Study on Applications of Room-Temperature Operating Silicon Single-Electron Transistors

室温動作シリコン単電子トランジスタとその応用

by

Kousuke Miyaji

宮地　幸祐

Submitted to the Department of Electronic Engineering
in Partial Fulfillment of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY
at the Graduate School of the UNIVERSITY OF TOKYO

December 17, 2007

Thesis Supervisor

Professor　Toshiro Hiramoto

Efficient Formal Equivalence Checking Methods for System-Level Design Descriptions

システムレベル設計記述に対する効率的な形式的等価性検証手法に関する研究

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 Electronics Engineering

Supervisor: Prof. Masahiro Fujita

Thesis submitted: December 13, 2007

Takeshi Matsumoto

Abstract

Due to the great advance of semiconductor technology, the integration of VLSI (Very Large Scale Integration) circuits has been increased for many years. This causes the size of designs of systems increases rapidly, which is a serious problem of the current VLSI designs. To reduce the design periods, system-level design is becoming widely accepted. System-level design is a design level where the execution order of the computation is decided, while the behavior of each clock cycle is precisely designed in register transfer level which is a starting point of conventional hardware design flows.

In system-level design, a lot of design changes are performed by designers' hand in order to refine and optimize designs. When a design bug inserted in system level is detected at the later design steps such as register transfer level or gate level, fixing the bug is much more difficult and time consuming than fixing a bug detected in system level. Thus, there is a great demand of equivalence checking of design descriptions between each refinement/optimization step.

One powerful method to check the equivalence is applying symbolic simulation to both of the designs under verification with generating equivalence classes of variables and expressions. This approach does not need any test patterns, hence, can be classified as formal verification. However, it cannot be applied to large designs since the run time of symbolic simulation increases exponentially to the design sizes. To realize efficient equivalence checking of large system-level design descriptions, in this thesis, several verification methods are proposed.

The proposed verification methods utilize the difference between the design descriptions. In practical, system-level design is proceeded by gradually refining designs step by step. Therefore, the difference between designs of one refinement step

is expected to be relatively small. The basic idea of the proposed method is that we can reduce the computation effort of equivalence checking utilizing the difference. Our proposed method can verify the equivalence by locally checking the equivalence of the difference and its related portions of designs. In this method, symbolic simulation is applied only to the small portions of the designs, which results in the significant reduction of the verification time.

When verifying designs including parallel behaviors, it is impossible to apply equivalence checking to all possible schedulings, since the number of schedulings increases exponentially to the design size. To solve the problem, a sequentialization method is proposed. Given a design description with parallel behaviors, the method generates an equivalent design description without parallel behaviors.

When symbolic simulation is applied to loops, they are unrolled to avoid the execution paths with infinite length. This results in the increase of verification time especially when the number of unrolling is large. As a solution of the problem, an equivalence checking method of loops without loop unrolling is proposed. It identifies the symbolic values of loop iterators that are required to compute an arbitrary index of the output arrays. After the required symbolic values of the iterators are extracted, symbolic simulation is applied only to the values for equivalence checking. As a result, the number of statements to be symbolically executed does not increase even if the number of iterations is actually large.

The several experiments conducted in this thesis confirm that the proposed methods enable to verify the equivalence of large system-level designs between a practical design refinement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 System-Level Design

Due to the great advance of semiconductor technology, the integration of VLSI (Very Large Scale Integration) circuits has been increased for many years. This enables to integrate more and more transistors on a chip, which results in that a large whole system can be realized as a single chip so called System-on-a-Chip (SoC). When we design an SoC, it is a serious problem that the design period tends to be very long since the design size of SoCs is much larger than that of the conventional VLSI designs. To use billions of transistors that can be integrated in a chip, the design productivity of SoCs should be much improved.

One solution to improve the design productivity is introducing system-level design as a starting point of SoC design. Figure 1.1 shows a typical flow of hardware designs. In the most hardware designs, designers start designing from RTL (Register Transfer Level). In RTL, the function executed by combinational circuits for each clock cycle and the hardware resources (for example, adder and multiplier) to execute the function are decided. On the other hand, in system level, the (partial) execution order of the behaviors and the functional units each of which consists of many memory elements and combinational circuits (for example, filter and inverse discrete cosine transform) to execute the behaviors are decided. Therefore, we can say that system level design is more abstract than RTL design in terms of time and hardware resources. Since SoCs usually implement a system consisting of both hardware and software, it is preferable to use a single design language to describe designs. To satisfy this need, C language or C-based design language is used in system level design, which enables to design both hardware and software seamlessly.

From the characteristics described above, the advantages of designing from system level with C-based language can be listed below.

- Faster simulation speed
- More flexible hardware/software partitioning
- More powerful optimization of architecture
- Less amount of design descriptions

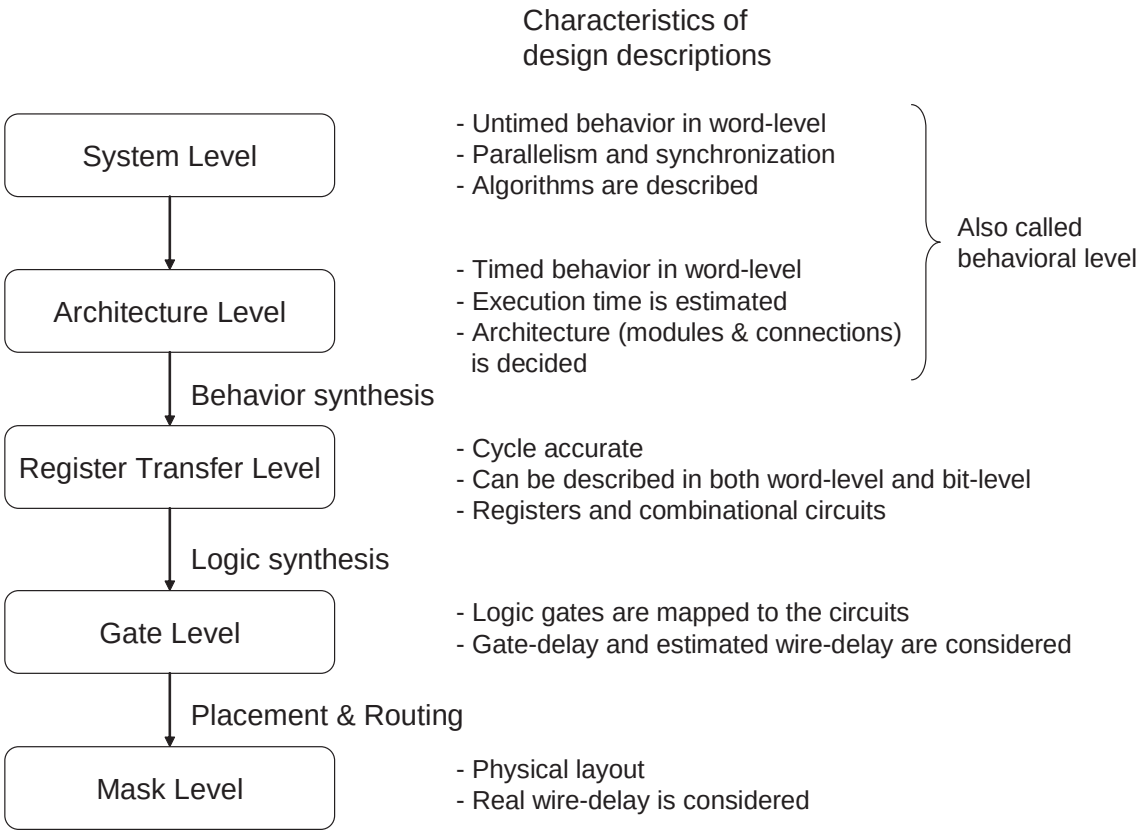


Figure 1.1: Hardware design flow

The first and last are coming from the high abstraction level of system level designs. The second and third are coming from hardware/software co-designs using C-based language. Due to the advantages above, in the last decade, system level design has been gradually accepted in industry, and many SoCs are designed from system level nowadays. In many C-based design language, SpecC [11] and SystemC [39] are widely used, since the standardization of them is actively proceeding. In addition, it is usual that a C-based design language and a development environment for the language are developed in companies.

1.2 Design Verification in System-Level

Generally, in each design step of SoC design, verification and debug to avoid bugs from designs are very important to ensure the correct behaviors of SoCs. However, they dominate 60% of the total SoC development period in average. In the case of an SoC design starting from system-level, verification in system-level is more important, since detecting and debugging bugs are more difficult when a bug that exists in system-level is detected in the later design stages such as RTL and gate-level. In those later stages, a design is described with more design descriptions, hence, designers have to read and understand large number of lines to detect and debug bugs. In addition, system-level design will be carried out again if any bug is detected in RTL or gate-level and cannot be fixed in there.

The facts above implies that efficient verification methods of system-level designs can significantly improve the design productivity. Therefore, in this thesis, verification methods of system-level design descriptions written in C-based language are focused. Figure 1.2 shows a typical flow of system-level design. Starting from a given specification, which is usually given as a C program, a design is gradually refined. For the hardware part, the design is refined down to behavioral descriptions that can be synthesized by behavioral synthesizers, then synthesized to an RTL design. For the software part, a C program that can work cooperatively with the hardware part is developed. In the most cases, this design process is done with C or C-based design description language. The following design refinements are applied in the flow.

- Software/hardware partitioning
- Algorithm optimization, especially for the hardware part. As the algorithms in original specifications in C language are not usually suitable for hardware execution, they should be optimized for hardware execution. For example, optimizations of loops are often carried out, which results in more efficient memory accesses and more parallel executions.
- Restricting the uses of syntax elements of C language that cannot be implemented in hardware, such as pointers, recursive callings, dynamic memory

allocations. These restrictions are coming from the fact that, different from in software, it is impossible to dynamically increase the computation resources in hardware.

- Introducing parallelism in the hardware part, and designing synchronization and communication among the parallel executions. Parallelization is one of the most important issues to improve the performance of hardware executions.

The design flow shown in Figure 1.2 is not completely automated. That is, before the final design descriptions that can be processed by behavioral synthesizers and compilers are generated, a number of refinements/changes/optimizations of design descriptions are carried out by designers themselves or by interactive design support tools. Therefore, it is very important to check the equivalence of the behaviors before and after the design descriptions are changed. This is because we can effectively detect bugs inserted during those design refinements above by applying equivalence checking since most of the refinements are not expected to change the behavior of the design. If those bugs are found in the later design steps, for example in RTL or in gate-level, a lot of time and cost will be spent to debug.

Based on the observations described in this section, in this thesis, equivalence checking of system-level design descriptions is studied. Checking the equivalence between a design under verification (DUV) and its golden model, which is sufficiently verified in advance, we can check whether or not there is any bug in the DUV. The proposed equivalence checking methods in this thesis are categorized as formal verification. Verification in system-level is mainly carried out by simulation. However, simulation based verification methods cannot guarantee the absence of bugs unless all possible input patterns/sequences are verified. Therefore, the quality of test patterns/sequences is very important in simulation. Different from simulation based methods, formal verification methods need not test patterns/sequences. Instead, the correctness of the design is proved by mathematical reasoning.

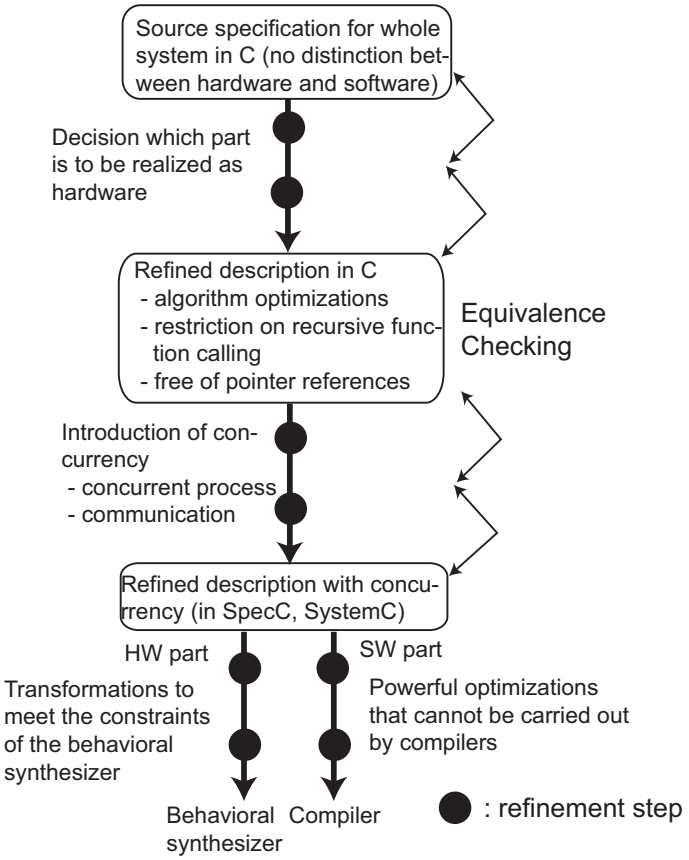


Figure 1.2: System-level design flow

1.3 Objectives and Target

Although, as described in the last section, formal verification enables verification without test patterns/sequences, the amount of the required computation for formal verification exponentially increases with the size of DUV. It results in that formal verification cannot solve the verification problem of large designs in practical time. The main objective of this thesis is developing formal equivalence checking methods that can solve the equivalence problem of large system-level design descriptions written in C-based language. When we assume the design flow shown in Figure 1.2, the difference between two consecutive descriptions of a single refinement step can be expected to be small. This is based on the observation that designs are gradually refined through the design flow. Therefore, it is possible to develop efficient equivalence checking methods utilizing the difference.

This thesis targets on system-level design descriptions that have the following characteristics.

- Can represent parallel executions and synchronizations
- Can represent passage of time
- Do not include pointer uses
- Do not include recursive function callings
- Do not include dynamic memory allocations

The design descriptions satisfying the characteristics above can be realized using C-based system-level design description languages such as SpecC or SystemC with some restrictions. The three restrictions listed above are come from the difficulties in implementing as hardware. Therefore, those restricted syntax elements do not usually appear in design descriptions of hardware part. On the other hand, they are usually included in descriptions of software part. As for pointer uses and dynamic memory allocation, pointer analysis techniques can remove them from design descriptions by identifying the location where a pointer can point. Recursive function callings can be removed when they are unrolled in advance of applying the proposed

verification methods. From the observation described here, after pointer analysis and loop unrolling are carried out as pre-process, the resulted descriptions can be verified with the proposed methods.

1.4 Contributions

In system-level design, there are few formal verification methods that are used widely in industry, and simulation plays a main role to verify designs. This is because large system-level designs cannot be solved by formal methods in practical periods. At the same time, however, simulation has also a serious problem on feeding good test patterns/sequences when a design is very large. Thus, formal methods and simulation should be applied complementary. To realize it, formal methods should be researched to increase the efficiency while simulation related methods is being researched. Currently, state-of-the-art formal verification methods can solve one module of system-level design, which is corresponding to the size that can be synthesized by state-of-the-art behavioral synthesizers. If our proposed methods can check the equivalence of two large design consisting of several modules, it can be said that the scalability of formal methods is much improved.

In this work, symbolic simulation, which is introduced in the later sections, plays a key role in the verification methods. Since the number of variables, expressions, and execution paths considered in symbolic simulation increase rapidly when the design size increases, it has a serious problem that the verification time increase exponentially to the design size. In this thesis, three novel verification methods to relax this problem. The key idea is that the difference between two target designs under verification is relatively small if we assume they are two designs before/after a refinement step in Figure 1.2. To utilize this fact, efficient verification methods are developed that can verify whole large designs.

1.5 Organization

This thesis is organized as follows:

In Chapter 2, fundamental techniques and related works are introduced. This

chapter introduces recent researches on formal verification of C-based descriptions, the overview and problems of symbolic simulation based methods, a previously proposed verification method for loop optimizations, source code refinements performed in system-level, and other fundamental technical issues.

In each chapter from Chapter 3 to Chapter 6, one proposed method of this work will be introduced one by one.

In Chapter 3, a sequentialization method that generates a functionally equivalent description having no parallel executions from the given original description with parallel executions and synchronizations. The method is useful when it is applied before symbolic simulation, since we do not need to consider interleaves among parallel executions any more after sequentialized.

In Chapter 4, an efficient method is introduced that reduces the number of equivalence checking between variables and expressions utilizing the difference between two designs. This method enables more efficient verification for designs with the same or similar control structures. However, the verification time by this method increases exponentially to the design size, although it can reduce a large number of equivalence checking between variables and expressions.

In Chapter 5, we propose a more efficient method utilizing the difference. In the method, symbolic simulation is applied only to the related portions to the difference, while it is applied from the start to the end of each path in the method in Chapter 4. This local checking approach results in that the equivalence can be proved with small computation effort even if the design itself is very large.

In Chapter 6, an equivalence checking method of two loops before/after loop optimization. This method targets on the situation that the design descriptions of loops with array accesses are refined so that the memory access patterns after behavioral synthesis can be better. The method does not unroll the loops under verification. Instead, it derives the numbers of the loop iteration to compute an arbitrary index of the output array. To check the equivalence of the output, symbolic simulation is just applied to generate the symbolic expression of the output arrays for an arbitrary index.

In Chapter 7, the proposed methods are evaluated with practical system-level designs. Two designs are prepared for this evaluation: one is MPEG4 decoder, the

other is an elevator controller. The results show that the proposed methods can solve the equivalence problem faster than the previous methods.

At last, in Chapter 8, the conclusions of this thesis are described with some further directions.

Chapter 2

Fundamental Techniques and Related Works

2.1 Equivalence Checking Methods for C-Based Design Descriptions

Several equivalence checking methods for C or C-based design descriptions have been proposed.

In [30], an equivalence checking method between an RTL design described in a subset of C language and one in Verilog is proposed. The design flow used in the work is shown as Figure 2.1. In the work, the design descriptions are originally described using a subset of C language. Then, they are translated into Verilog with some optimizations to improve the performance and synthesized to netlists. One of the advantages of this design methodology is that simulation can be performed faster since the original descriptions are in C language. In the flow, equivalence checking should be applied in the following two cases: between RTL C and RTL Verilog and between RTL Verilog and Netlist. The latter can be accomplished with some commercial tools already used in industry. For the former case, they have developed a translator from RTL C to Verilog and formally verified the equivalence between the translated Verilog description and the optimized. This is also accomplished by existing equivalence checkers for RTL designs. Different from this work, we have tried to develop equivalence checking for system-level designs since system-level design with C-based languages has been gradually spread in industry in these years.

In [4, 34], equivalence checking between designs before and after behavioral synthesis is proposed. The designs before synthesis are in C-based language and ones after synthesis are in RTL HDL such as Verilog and VHDL. In the method, both of the designs under verification are translated in FSMD (Finite State Machine with Data). By making correspondences between the two FSMDs, the equivalence is verified. To reduce the computational effort to make the correspondences among the states in the FSMDs, the method uses the information from the behavioral synthesis tool. Otherwise, it is almost impossible to check the equivalence of the large designs in practical time.

In [18], an equivalence checking method for two designs in FSMD is proposed. The objective of the work is checking the equivalence of two FSMDs before and after

scheduling is changed. In the work, utilizing the similarity between two FSMDs, cutpoints are inserted to reduce the enumeration of the execution paths.

The equivalence checking methods described above target the equivalence checking of two RTL designs before and after optimizations, or between a behavior-level design before behavioral synthesis and an RTL design synthesized from it. Different from those works, our target is equivalence checking between two behavior-level designs.

In equivalence checking of large designs, it is essential to make correspondence between two designs, which can significantly reduce the space to be checked. Suppose the equivalence is checked for each pair of execution paths from two designs. Since there are a large number of execution paths when the designs are large, it is impossible to check the all combinations of them. If the correspondence is successfully taken, the number of paths to be checked are significantly reduced. To utilize the correspondence between two designs, the data flows should be similar in [30], and the information on the state correspondence from behavioral synthesizer is used in [34]. Compared to those methods, our methods utilizes the difference between designs. This approach is expected to work well since we target the equivalence checking between two designs before and after one refinement step in Figure 1.2.

An equivalence checking method of two behavior-level designs is proposed in [7, 8]. This method targets designs described in ANSI-C language and does not assume the large similarity or little difference between two designs. It first converts the given designs into bit-vector equations, where each variable represents a bit of integers in the original C descriptions. Then, the equations are converted to boolean satisfiability problem to check the equivalence, which is solved by SAT solvers such as chaff[22]. The method is implemented and published as a tool CBMC[41]. One of the most important advantages of CBMC is that it supports all syntax elements of ANSI-C. Since CBMC applies the conversion to the whole of the design descriptions without any abstraction, it is usual that it cannot verify the equivalence of the designs with hundreds of lines in practical time.

One limitation of the method in [7, 8] is that the resulting SAT problem is too large. To overcome this problem, three possible solutions are listed below.

- Apply word-level reasoning techniques instead of bit-level ones

- Verify the part of the design instead of the whole
- Apply abstraction

As proposed in the later chapter, the second is realized by analyzing the design descriptions locally around the difference. For the first item, word-level symbolic simulation, that is introduced in the next subsection, can be a solution. Applying it, the number of variables in formulas to be solved for the equivalence can be reduced to less than one tenth, which results in the improvement of the verification speed. In this work, abstraction is not considered.

An equivalence checking method of two behavior-level designs in C-based language is proposed in [24]. In this work, the method is based on word-level symbolic simulation, hence, it can verify relatively large designs compared to the methods introduced above. However, even the method cannot verify a large design since it is impossible to perform symbolic simulation of the whole of large designs. Based on the work in [24], we propose more efficient verification methods utilizing difference between design descriptions.

In industry, there are few available commercial tools to check the equivalence of system-level designs. Of them, SLEC from Calypto Inc. is one of the most popular tools in this field. Even using SLEC, the design size that can be verified in practical time is limited to one module of a whole system-level design consisting of tens of modules. Therefore, the goal of this work is to achieve the equivalence checking of large system-level designs in practical time.

2.2 Symbolic Simulation Based Formal Equivalence Checking

Symbolic simulation is a simulation technique where each value of a variable and arithmetic operation among variables is treated as a symbol. A symbol corresponding to a variable can represent all possible values of the variable. Therefore, the resulting expression from symbolic simulation corresponds to the result of simulating all possible input patterns, hence, symbolic simulation can be applied as a formal method. Currently, many formal verification methods based on symbolic simulation

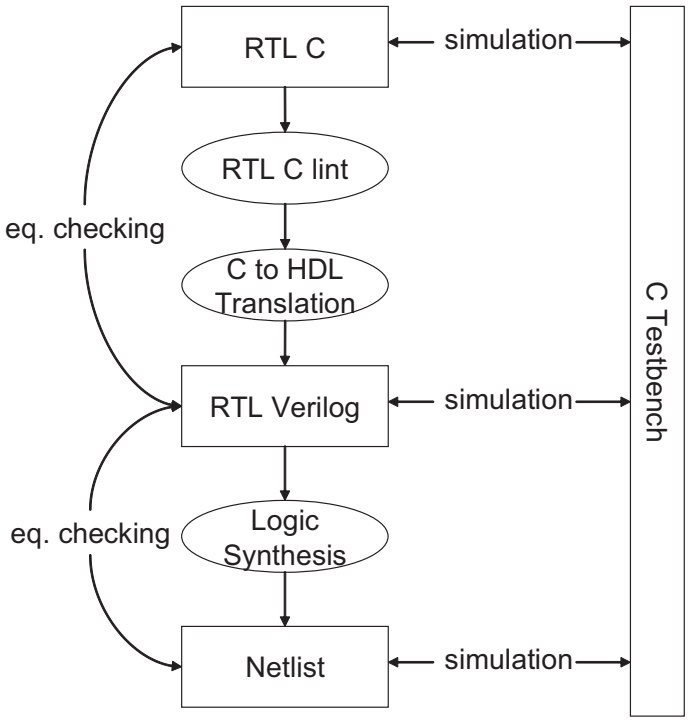


Figure 2.1: Design and Verification flow proposed in [30]

are proposed and applied to practical designs [4, 9, 25, 34].

Symbolic simulation based equivalence checking is first proposed in [24] for RTL and gate-level designs. In the method, Equivalence Class (EqvClass) plays an important role to check the equivalence. Every variable and expression in an EqvClass is equivalent to each other. In the method in [25], by repeating to generate and merge EqvClasses during symbolic simulation, the equivalences of registers and memories are checked.

In this thesis, as the target is system-level design descriptions written in C-based language, we have developed our own symbolic simulator for the purpose, which will be described in the later chapters. It symbolically executes two given C-based design descriptions with generating and merging EqvClasses. Before staring symbolic simulation, the input variables and the output variables are specified. Based on this specification, symbolic simulation is carried out assuming the corresponding input variables from the two designs under verification are equivalent to each other. If the

pair of symbolic expressions corresponding to the output variables are all proved to be equivalent, the verification result is "equivalent".

The fundamental procedures of our developed symbolic simulator can be described as follows:

- Two designs to be symbolically executed, the correspondences of the input variables and the output variables between the designs are given.
- When starting symbolic simulation, each pair of the corresponding input variables are assumed to be equivalent, and set them to the same EqvClass.
- When an assignment statement is executed, the variable of the left-hand side and the expression (or variable) of the right-hand side are put into the same EqvClass.
- If two variables/expressions in different EqvClasses are proved to be equivalent during symbolic simulation, the two EqvClasses are merged into a single EqvClass.
- When a conditional branch is executed, only a path where the execution condition is satisfied is taken. If the condition cannot be evaluated to be true or false, both of the two paths are executed one by one. It means that symbolic simulation is applied to all possible execution paths in the designs.
- When finishing symbolic simulation of each pair of two execution paths from the two designs, the equivalence of all pairs of the output variables are checked. At this time, if any result of a path is not proved to be equivalent, we can conclude that the final result is "not proved to be equivalent."
- After simulating all possible paths, the final result is "equivalent," if the results of the execution paths are all equivalent.

To distinguish the symbols before and after an assignment, the name of the symbols must be changed at every assignment. This is realized by adding suffix to the name of the symbols for variables. In the same path, we can distinguish

a variables v of different number of assignments by naming it with v_n, where n denotes a number of assignments for the variable v.

When function calls exist in design descriptions, there are two ways to apply symbolic simulation. One way is simulating a called function by entering inside the function, which the same way as what simulation usually does. The other way is treating a called function as uninterpreted. Two calls of uninterpreted function are equivalent just if the called functions are the same and all pairs of corresponding parameters are equivalent. Therefore, considering functions as uninterpreted can significantly reduce the computational effort of the verification.

A simple example of equivalence checking in terms of symbolic simulation is shown in Figure 2.2. In this example, we verify the equivalence of the variable $reg0$ in the two given descriptions. Initially, the variables $reg1$ and $reg2$ are assumed to be equivalent in both descriptions, because these variables correspond to input signals. These assumptions are expressed in the two EqvClasses E1 and E2. Note that we denote a variable v in description 1 as v_1 and in description 2 as v_2.

At first, expressions for the variables $src1$ and $src2$ in description 2 are simulated before reaching point (A). This results in that $src1\_2$ is inserted into E1 and $src2\_2$ is into E2, because $src1\_2$ is equal to $reg1\_2$ and $src2\_2$ is equal to $reg2\_2$. Then, two additional EqvClasses E3 and E4 are created before reaching point (B). Finally, $reg1\_1$ and $reg2\_1$ are substituted for $src1\_2$ and $src2\_2$ in E4, respectively, because from E1 and E2 we can find out that $src1\_2$ is equivalent to $reg1\_1$ and $src2\_2$ is equivalent to $reg2\_1$. This results in that E3 and E4 are equivalent to each other. Therefore, E3 and E4 are merged into a new EqvClass E3'. As a result, we can conclude that the variable $reg0$ are equivalent in both descriptions, because $reg0$ in both descriptions are in the same EqvClass.

2.3 Equivalence Checking for Loop Optimizations

Recently, in [31], Shashidhar et al proposed an equivalence checking method for transformations on array-intensive source code. They construct two Array Data Dependence Graphs (ADDGs) for two loops under verification. At the same time, the index relations between arrays in define-use relation are analyzed and represented

as maps. An example loop code and the corresponding ADDG are shown in Figure 2.3. The equivalence of two loops can be verified by checking that every path in ADDGs has an identical data flow and input-output index mapping. In the method, the program codes must be:

- Single assignment form (every array elements are assigned once)
- Static control-flow
- Affine indexes
- No pointer references

Otherwise, the codes have to be transformed to this class.

Later chapters will show that our method can accept the codes with less restrictions. That is, we can handle:

- Not only arrays but also variables, and they can be assigned multiple times
- Non-static control-flow, which is usually introduced for short-cut paths or rounding up/down of data
- Extra codes outside loops that are added during loop optimization, which are typically added to realize pre- and post-process of pipelining optimization

Also, the verification algorithm is different from that in [31]. In the method in [31], basically, the elements of the output arrays to be checked the equivalence have to have the same expressions. On the other hand, our method employs symbolic simulation based method with equivalence rules and decision procedure, as described in the following chapters.

2.4 Code Refinements in System-Level Design

When we consider equivalence checking in system-level, it is essential to know what refinements are carried out on designs to improve the performance. In [32], a code refinement methodology in system-level design is introduced. One of the motivations of manual code refinements in system-level is that behavioral synthesizers cannot

perform all optimizations that can be potentially applied to the designs. This is because behavioral synthesizers cannot recognize candidates of optimizations depending on the coding style. To explicitly show the candidates of optimizations and improve the application of those optimizations, several refinement rules are proposed in [32].

- **Conversion to constant.** When an input parameter of a function is only read and not assigned in the function, the parameter should be declared as a constant. It improves the flexibility of the memory accesses for the parameter variable.
- **Conversion to explicit data flow.** The common uses of global variables causes problems in synthesis. By explicitly declaring as an input parameter of a function, the function calls can be scheduled into parallel executions.
- **Conversion to fixed point.** Conversion of floating point arithmetic operations to fixed point can significantly reduce the hardware area and performance.
- **Conversion to explicit memory accesses.** Pointers pointing to array elements prevent behavioral synthesizers from scheduling the array accesses in flexible. Removing pointers for arrays, the synthesizers can perform an optimized scheduling of array accesses.
- **Constant input enumeration.** When a function is written assuming an input parameter has a small number of specific values, the function can be optimized based on the values. By declaring enumerations of such input parameters, behavioral synthesizers can recognize the opportunities of optimization.
- **Loop rerolling.** Loop unrolling is usually applied to optimize software programs. However, hardwares synthesized from unrolled loops sometimes result in begin larger area and less performance. To avoid this, for hardware parts, loops should be rerolled.
- **Conversion to explicit control flow.** In this guideline, function pointers are removed to explicitly tell the control flow to behavioral synthesizers.

- **Function specialization.** When a parameter for a function has a frequent value, the function can be optimized much by considering the value as a constant.

- **Algorithmic specialization.** Replacing a software algorithm with an algorithm that is more appropriate for hardware possibly achieves order of magnitude improvements.

- **Pass-by-value return.** If an array and a pointer are input parameters of a function and the pointer never points to any of the array elements. The data of the array can be prefetched before it is actually read, regardless the location the pointer points. To realize this by behavioral synthesizers, the array should be copied to a local array at the beginning of the function, and write the data of the local array back to the original at the end of the function. It explicitly tells synthesizers that the local array is independent from the location pointed by the pointer.

Those code refinements are applied to improve to apply more optimizations by behavioral synthesizers. Thus, if the equivalence checking method proposed in this thesis can verify the refinements, it can be used in practical system-level designs.

Another type of code refinements in system-level design is removing pointers. Since ANSI-C allows to use pointer variables to point variables, arrays, and functions, it is possible that a system-level design description includes such pointer uses. However, although there are some works to synthesize pointers to hardware[28, 29], even state-of-the-art behavioral synthesizers cannot synthesize pointers in the most cases. Therefore, in system-level design, the removal of pointers is an important refinement.

Some refinements can be carried out automatically in the process of behavioral synthesis. Some of such refinements are implemented as a synthesis tool SPARK[14]. According to the work, code motion, speculative code motion, dynamic renaming, and dynamic common sub-expression elimination (CSE) are implemented to improve the performance of synthesized hardware. Those automatic refinements are also the target of this thesis, since equivalence checking for the refinements is still essential to guarantee the correctness even if they can be carried out automatically.

Design refinements to map a given system to a set of processing elements are proposed in [1, 2]. In those works, a system is modeled by a set of behaviors. Then, the behaviors are assigned to processing elements in deciding the architecture of the system. Basically, the processing elements run concurrently. At the same time, channels for communication among the processing elements and synchronizations are added to the system. During the refinements, the behaviors themselves are not changed. This process to map behaviors to an architecture is supported by the tool SCE [36].

In this thesis, the main target of equivalence checking is the code refinements of assignment and branching statements. Thus, rewriting only the declarations is not considered in this thesis. Also, since we assume that the designs under verification have no pointer, the code refinements to pointer variables are not considered. The proposed methods can be used to prove the equivalence of two designs that are obtained by both manual and automatic code refinements. Therefore, the automatic code refinements, for example in SPARK[14], can be included in the target of this thesis. In practical, since the cost of bug fixing is significant in hardware designs, it is desirable to verify the equivalence when a design is refined regardless manually or automatically.

2.5 System Dependence Graph

To express C-based design descriptions or C programs, many data structures have been proposed. Of them, System Dependence Graph (SDG) is widely used in debugging and analyzing design descriptions. SDG of programs including function calls is originally proposed in [17]. One of the advantages to use the SDG in [17] is that we can traverse inter-procedural (inter-function) dependence paths faster, since it has summary edges to summarize the dependence between the input parameters and output parameters of a function, which results in we save the traverse of the dependence inside the function.

Figure 2.5 shows an example SDG. In SDGs, each node represents a statement and each edge represents a dependence between two nodes. There are several kinds of dependence edges. In the figure, solid lines represent data dependences, while

dotted lines represent control dependences. A data dependence from a node n_1 to another node n_2 exists, when a variable v is defined at n_1 and used at n_2. On the other hand, a control dependence from a node n_1 to another node n_2 exists, when the execution of n_2 is controled (in other words, guarded) by n_1. Therefore, when a node correspondingly to if appears in a design description, every statement node under the if node are connected with a control dependence edge.

In this work, Extended System Dependence Graph (ExSDG) is used to check equivalence efficiently. ExSDG is proposed as an extension of SDG in [23]. As the original SDG is used mainly for program slicing[35], each node needs to have only a line number of the corresponding source code. An example of commercially program slicer is CodeSurfer[37], which can construct an SDG and perform slicing for a large C/C++ programs. On the other hand, to use SDG for verification and debugging, each node must have syntax and semantic information. In ExSDG, such information is kept as a form of Abstract Syntax Tree (AST). Thus, we can find useful information from nodes in ExSDG, compared to the case each node in SDG just shows the line number in the source code. ExSDG has the following advantages:

- The syntax of AST in ExSDG is simple enough to verify, but also extended ANSI-C enough to express hardware designs in both behavioral level and RTL. The syntax is free from language specific problems.

- AST and SDG are tightly integrated in ExSDG. For example, some nodes of the AST are shared with SDG. It enables verification tool developers to analyze design easily.

- The number of nodes and edges is less than that of existing SDGs, since the SDG in ExSDG is highly optimized. Then, the verification time of methods using ExSDG is expected to be shorter.

Figure 2.6 shows an example ExSDG for an example C-based design description in Figure 2.5. The shadow nodes are nodes of SDG and the other nodes are of AST. The nodes whose type is Function, Statement, or Variable are shared between AST and SDG. As shown in the figure, the nodes of AST and SDG are tightly integrated, which enables to perform efficient verification and analysis.

2.6 Validity/Satisfiability Checking of Logic Formula

In symbolic simulation, the equivalence of the following pairs of expressions cannot be proved, because symbolic simulation does not interpret the functionality of the expressions.

$$a + a,\ 2 * a$$
$$(a + b) + c,\ a + (b + c)$$
$$a * (b + c),\ a * b + a * c$$

To prove the equivalence of these expressions, commutative law and distributive law of arithmetic operations must be considered.

To prove such equivalences that cannot be proved by symbolic simulation, our proposed methods utilize decision procedures of formulas. A decision procedure is a tool that can decide the validity/satisfiability of a given formulas. Solving boolean satisfiability problems by SAT solver such as Chaff[22], we can decide the validity/satisfiability of a propositional formula. However, it is more difficult to solve the validity/satisfiability of a formula including arithmetics. Recently, due to the advances in the researches on Satisfiability Modulo Theory solver[12, 10], the validity/satisfiability of formulas with arithmetics can be solved faster.

In the proposed methods in this thesis, a decision procedure Cooperating Validity Checker[40] is utilized. CVC is a decision procedure that checks logical validity or satisfiability of given formulas [33]. Formulas are represented by propositional operators and equations between linear mathematical operators. CVC can basically accept quantifier-free formulas in first order logic. In addition, the formulas can have the followings:

- linear real arithmetic formulas. The supported operators are addition, subtraction, multiplication by a constant, division by a constant, equality, and inequality
- real arrays
- inductive data types (for example, lists and trees)

By using CVC in our methods, the ability of equivalence checking between variables is improved. Compared to substitution in symbolic simulation, generally, CVC takes longer time to compute equivalence because it utilizes several theorems to check validity.

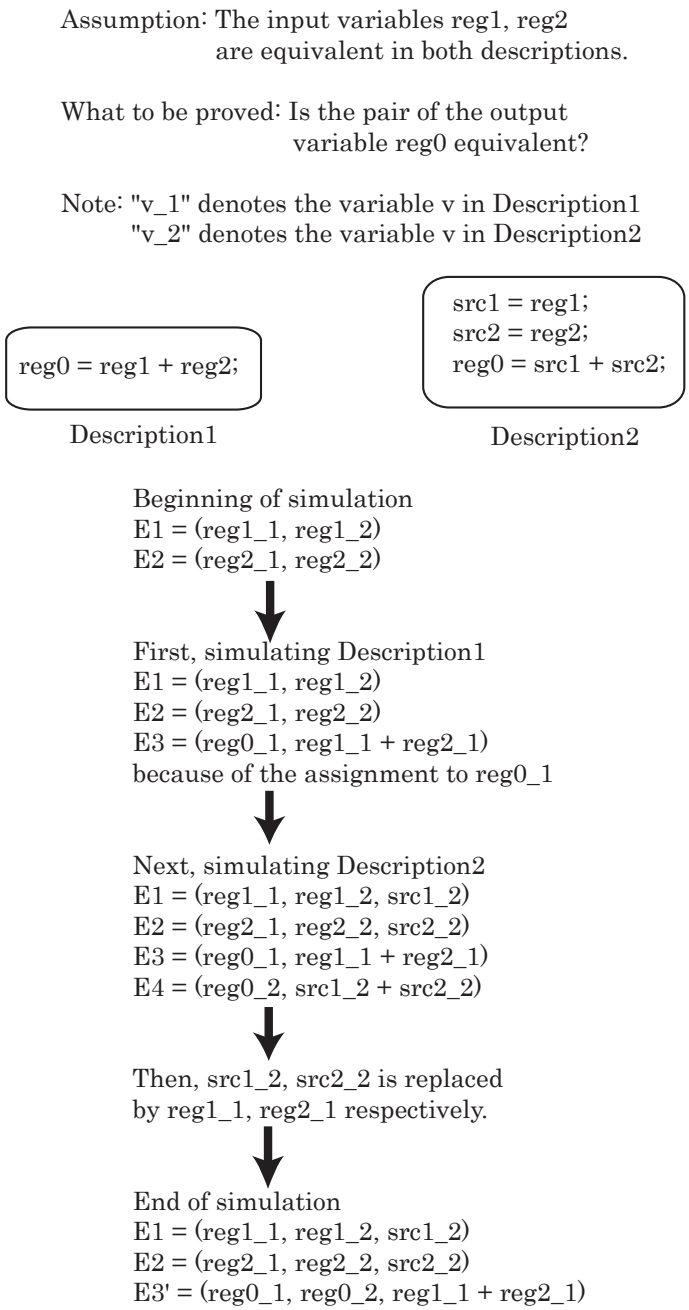


Figure 2.2: An example of equivalence checking based on symbolic simulation

```
#define N 1024
foo(int A[], int B[], int C[]) {
  int k, tmp[N], buf[2*N];
  for(k=0; k<N; k++)
s1:  tmp[k] = B[2*k] + B[k];
  for(k=N; k>=1; k--)
s2:  buf[2*k-2] = A[2*k-2] + A[k-1];
  for(k=0; k<N; k++)
s3:  C[k] = tmp[k] + buf[2*k];
}
```

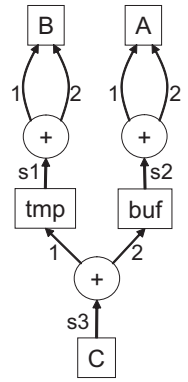


Figure 2.3: An example loop and its ADDG[31]

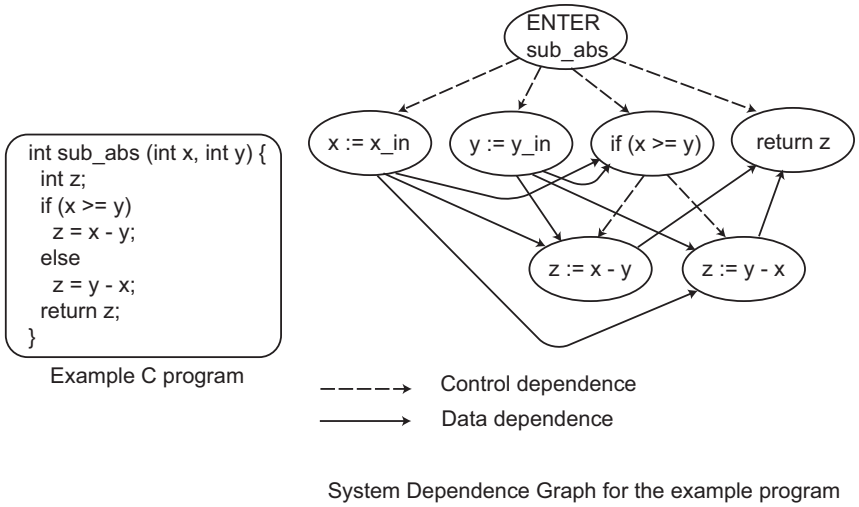


Figure 2.4: An example of System Dependence Graph

```
module Divider(i1, i2, o1, o2, rst, done, clk);
  input [7:0] i1, i2;
  input rst, clk;
  output reg, done;
  reg [7:0] x;
  always@(posedge clk)
  begin
    if(rst) begin
      x <= i2;
      o1 <= 1'b0;
      o2 <= i1;
      done <= 1'b0;
    end
    else if(o2 >= x) begin
      o2 <= o2 – x;
      o1 <= o1 + 1;
    end
    else
      done <= 1'b1;
  end
endmodule
```

(a) Divider in Verilog-HDL

```
module Divider(bit[8] i1, bit[8] i2, bit[1] rst,
                buffered bit[1] done) {
  buffered bit[8] x;
  void main() {
    while(1) {
      if(rst) {
        x = i2;
        o1 = o;
        o2 = i1;
        done = 0;
      }
      else if(o2 >= x) {
        o2 = o2 – x;
        o1 = o1 + 1;
      }
      else
        done = 1;
      waitfor(1);
    }
  }
};
```

(b) Divider in C-based Language

Figure 2.5: Divider RTL designs written in (a) Verilog-HDL and (b) C-based language

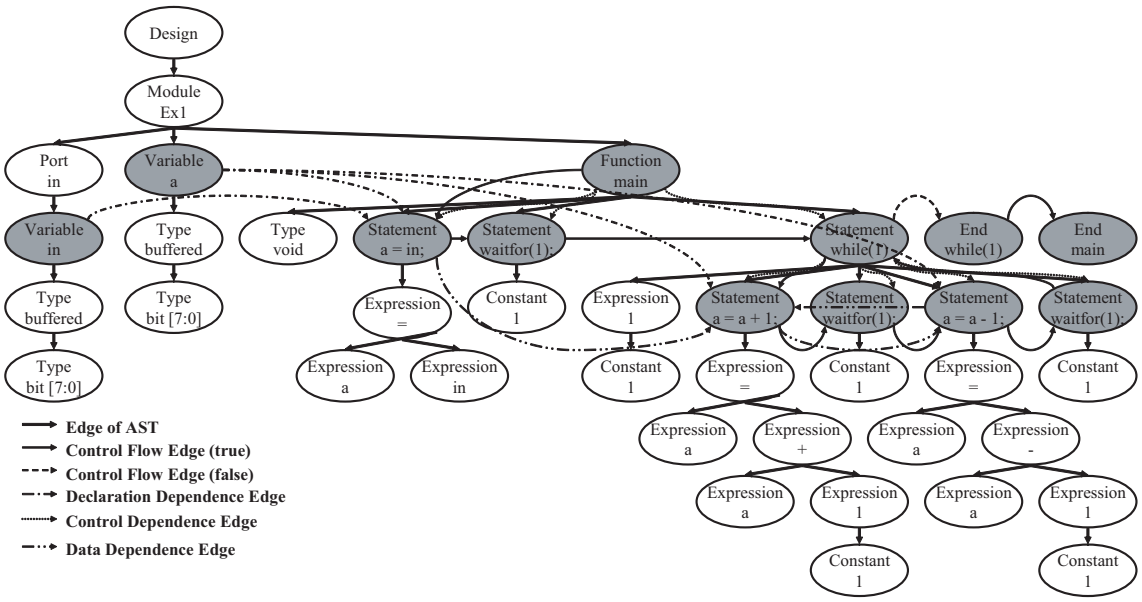


Figure 2.6: ExSDG of the design shown in Figure 2.5

Chapter 3

Sequentialization Method of Parallel Behaviors

3.1 Introduction

Generally, statements in parallel behaviors are not executed deterministically without correct synchronization. This is because the execution order cannot be decided to be one order, which results in the values of the variables computed in the parallel behaviors are different among different execution orders. When considering equivalence checking of such parallel behaviors, we have to check all possible execution orders. However, since the number of the possible execution orders increases exponentially with the number of statements in parallel behaviors, checking equivalence for all possible orders are not practical. Partial order reduction can be solved this problem. It reduces the execution orders whose results become the same as another. Applying partial order reduction, although we can identify a set of execution orders that represents all possible orders, to decide the equivalence of the results generated by different execution orders, equivalence checking need to be carried out in the process of partial order reduction.

In this chapter, we propose a method to sequentialize parallel behaviors with dependence and synchronization. The method consists of the following two steps: (1) deciding whether or not a given design including parallel executions can be sequentialized, (2) doing sequentialization based on the results of (1). In the method, it is important that the resulting sequentialized design is equivalent to the original parallel design for all possible execution orders. To guarantee it, the method checks that two statements having dependency are always executed in the same order. This can be solved by using ILP solvers. As this check is required only for the statements that are executed in parallel and have dependency, the method is expected to be able to sequentialize large designs.

3.2 Synchronization Verification Using ILP Solver

Software model checking poses its own challenges, as software tends to be less structured than hardware. In the software verification domain, predicate abstraction [13, 15] is widely applied to reduce the state-space by mapping an infinite state-space program to an abstract program of Boolean type while preserving the behaviors and

control constructs of the original. Counterexample-Guided Abstraction Refinement (CEGAR) [6] is a method to automate the abstraction refinement process. More specifically, starting with a coarse level of abstraction, the given property is verified. A counterexample is given when the property does not hold. If this counterexample turns out to be spurious, the previous abstract programs are then refined to a finer level of abstraction. The verification process is continued until there is no error found or there is no solution for the given property.

In system-level design languages such as SpecC, extra constructs are added to C in order to describe the characteristics of hardware. These extra constructs support description of parallel behaviors, pipelined behaviors, finite state machines, and operations on arbitrary-length bit-vectors. System-level models are organized as a collection of cooperating processes running in parallel. In order to keep all processes executing as the designer intended, proper scheduling of statement execution in all processes (known as synchronization) is necessary. Deadlock is an error that is caused by synchronization failure.

In [26], an approach to synchronization verification of systems described in SpecC was proposed. SpecC contains the $waitfor$ and $notify/wait$ statements to schedule and synchronize concurrent processes. The $waitfor$ statement delays a process by a specific number of time units and therefore introduces a timing constraint. In this approach timing constraints are expressed with equalities/inequalities which can be solved by integer linear programming (ILP) solvers.

Figure 3.1 (b) shows an example of timing constraint of a design in SpecC described as 3.1 (a). The two behaviors under a *par* statement start at the same time and finish at the same time, respectively. A *wait* statement suspends the execution of the behavior that the *wait* is included, until one of the corresponding $notify$ statements is executed. Therefore, the execution times of a *wait* statement is later than that of the corresponding $notify$.

From the timing constraints from *par* and $wait/notify$, we can derive the inequality equations as shown as Figure 3.1 (c). In the formulas, T_{start_X} and T_{end_X} mean the starting time and ending time of a statement/behavior X, respectively. In addition, T_{notify} corresponds to the time when the $notify$ statement is executed, while T_{wait} corresponds to the time when the *wait* statement is notified by $notify$.

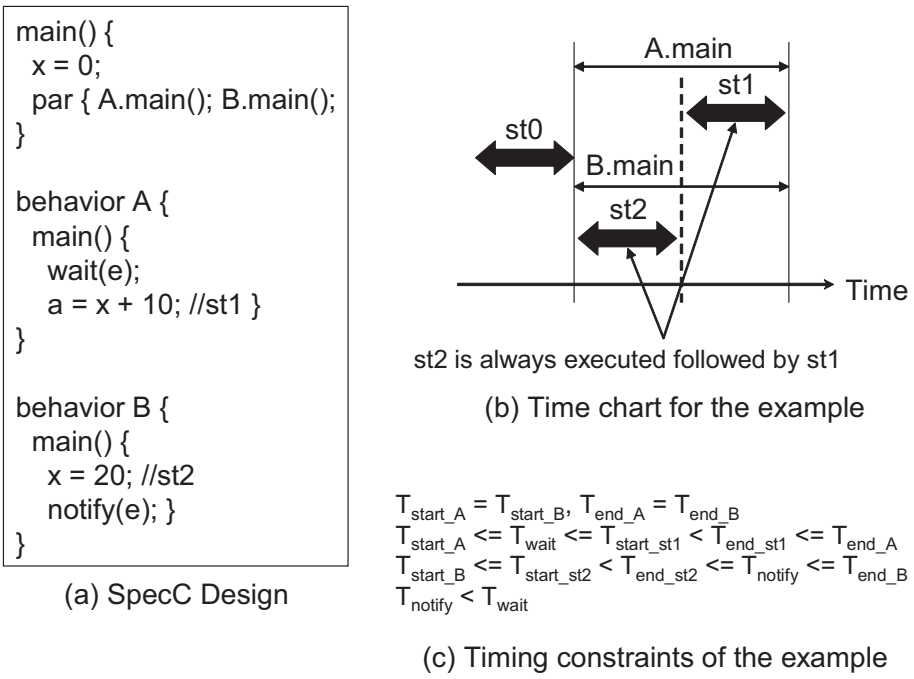


Figure 3.1: Example of synchronization checking

Adding the property describing the proper timing between parallel behaviors, ILP solvers can solve whether or not the property is always satisfied. For example, when checking that no deadlock in the design, the property can be described as $(T_{notify} - T_{wait} < 0)$ which means the *wait* statement is always followed by the $notify$. If the property fails (for example, if deadlock exists), ILP solvers cannot find a set of assignments to time variables since there is some conflict among the given timing constraints.

In the proposed sequentialization method, as described from the next section, the possibility of sequentialization is checked by solving the property describing that the execution order of parallel statements do not change under the timing constraints generated from the design.

3.3 Proposed Method

To sequentialize the design descriptions, the following conditions must be satisfied.

- **No deadlock.** Every *wait* statement is always executed with at least one corresponding *notify* statement being eventually executed.
- **No race condition.** There is no possibility that any shared variable to be accessed by two or more parallel behaviors generates the different results.

These are the necessary conditions required to generate the sequential design that is equivalent to the original one. If a given design has a deadlock, its execution can halt somewhere in the design, while the execution of the sequential design should never halt. In this case, behaviors of the two designs are not equivalent for any sequentialization. On the second condition, if there exists any global variable which is local to parallel behaviors, access to this variable (known as read/write or write/write access) at the same time can cause the design to perform different functionalities.

The first condition can be solved by the verification method in [26]. To check the second condition, the following two cases should be considered.

- For a variable shared by two or more parallel behaviors, every two assignment statements to the variable is always executed in the same order (to guarantee the same execution order for a write-write dependence).
- For a variable shared by two or more parallel behaviors, an assignment statement to the variable and a statement using the variable are executed in the same order (to guarantee the same execution order for a read-write dependence).

Note that the results of the parallel behaviors can be equivalent if the two conditions above are not satisfied. For example, even when assignment statements to a shared variable are executed in parallel without any synchronization, the results are always equivalent if the assigned values are the same constant. In this work, however, those cases are not considered to be sequentializable, since it is better to conclude that there is some errors in synchronization in most of those cases.

Before introducing race condition check, definitions of basic block and synchronization point are introduced.

- **Basic block (BB):** A series of statements that does not include conditional branches nor synchronization point.

- **Synchronization point (SP):** A pair of $notify$ and $wait$ statements of the same event.

Now, assume that there are two basic blocks, $BB1$ and $BB2$, where $T(BB1_{starttime}) < T(BB1_{endtime})$ and $T(BB2_{starttime}) < T(BB2_{endtime})$ are the timing constraints for each block. Together with these constraints, we can check the race condition between $BB1$ and $BB2$ by checking the following pair of properties.

$$\textbf{Prop1}: \ T(BB1_{starttime}) < T(BB2_{endtime})$$
$$\textbf{Prop2}: \ T(BB1_{endtime}) > T(BB2_{starttime})$$

We will not consider the condition when both **Prop1** and **Prop2** are unsatisfied at the same time since it is obvious that this condition cannot be occurred. If **Prop1** is satisfied and **Prop2** is not, $BB1$ is proved to be executed prior to $BB2$ ($BB1 \rightarrow BB2$). In contrast, if **Prop1** is not satisfied and **Prop2** is, we can say that $BB1$ is proved to be executed after $BB2$ ($BB2 \rightarrow BB1$). If both of the properties are satisfied, *BB1* and *BB2* can be interleaved and it is possible for race condition to occur. All variables in *BB1* and *BB2* must be checked for data dependency. If, for any variable, there is data dependence (read/write or write/write) between *BB1* and *BB2*, then there is a race condition. Otherwise, *BB1* and *BB2* are race condition free and we can sequentialize in either ($BB1 \rightarrow BB2$) or ($BB2 \rightarrow BB1$) order.

According to Algorithm 1, the sequentialization process is described as follows.

- Since SpecC supports design hierarchy and concurrency, we store all *par* statements in a set SET_{par} and, for each *par*, we mark its depth as we explore the design to lower hierarchy. The depth of a *par* statement is the depth from the top *par* statement of the hierarchy. If a *par* statement is not nested, the depth is 0.

- For each *par* statement, the deepest depth of SET_{par} is taken from the set and sequentialized by the following procedure. Let $Bhvr1$ and $Bhvr2$, for example, denote the two behaviors under the *par* statement to be sequentialized.

Algorithm 1 Sequentialization(SC)

```
declare
 1:  error: a Boolean variable
 2:  Sync: a set of synchronization point
 3:  SET_par: a set contains heuristic depth of par
begin
 4:  /* Heuristically search for par */
 5:  SET_par := HeuristicSearch(SC)
 6:  foreach par in SET_par (start from the deepest one) do
 7:    /* Check if there is any deadlock */
 8:    (error, Sync) := SynchronizationVerification(SC)
 9:    if error then
10:      exit ("There is a deadlock")
11:    end if
12:    foreach synchronization point in Sync do
13:      /* Check if there is any race condition */
14:      error := RaceConditionDetect(SC)
15:      if error then
16:        exit ("There is a race condition")
17:      end if
18:    end for
19:  end for
20:  return (SequentialGen(SC))
end
```

- With the method for race condition check as described above, find all pairs of basic blocks $BB1$ in $Bhvr1$ and $BB2$ in $Bhvr2$ and put them into a set BB_{pair}.
- For each pair in BB_{pair}, check **Prop1** and **Prop2**. If **Prop1** is true and **Prop2** is not, *BB1* and *BB2* can be sequentialized directly as *BB1* before *BB2*, and vice versa. However, if both properties are satisfied and there is no data dependence, *BB1* and *BB2* can be sequentialized in any order. Either *BB1* before *BB2* or *BB2* before *BB1*.
- Otherwise, race condition is found and verification terminates with error report.

Examples

Figure 3.2 shows three examples of sequentialization. In each example description, there are two parallel behaviors. In the following, the possibility of sequentialization and the result of sequentialization if possible are explained one by one.

In the example shown in Figure 3.2 (a), there is no synchronization in both of the behaviors. Also, there is no dependence between the behaviors. Therefore, we can sequentialize them in any order without race condition check, as long as the order in each behavior is not changed.

The example in Figure 3.2 (b) has a pair of $notify$ and $wait$ statements, hence, they are synchronized at those points. In the behaviors, a shared variable x is used in right-hand side behavior and defined in left-hand side. Therefore, the execution order of $c = a + x$ and $x = 20$ has to be checked. As a result, due to the synchronization, the order is always same. That is, $c = a + x$ is always executed after the execution of $x = 20$. To satisfy the order between the two statements, they are sequentialized as shown in the figure.

In the example of Figure 3.2 (c), although there are dependences related to a variable x, no synchronization is performed. For example, $x = 10$ and $x = 20$ are included in the parallel behaviors, but the execution order cannot be decided to be one. In such cases, we do not sequentialize behaviors nor apply formal equivalence checking methods described in this thesis.

3.4 Experimental Results

We have experimented our tool on several designs written in SpecC. Since we focus on proving mainly for correctness of hardware, our tool does not support the use of pointers, dynamic memory allocation, and recursive functions. If the design under verification contains any of these, we need to remove them manually. Fortunately, in our experiment, all designs contain none of them.

For the experiments, we have three sets of SpecC design descriptions of Inverse Discrete Cosine Transform (IDCT), Vocoder, and MP3 decoder. The original descriptions are refined by introducing parallelism and changing the scheduling. Those refinements are frequently carried out in system-level to improve the performance

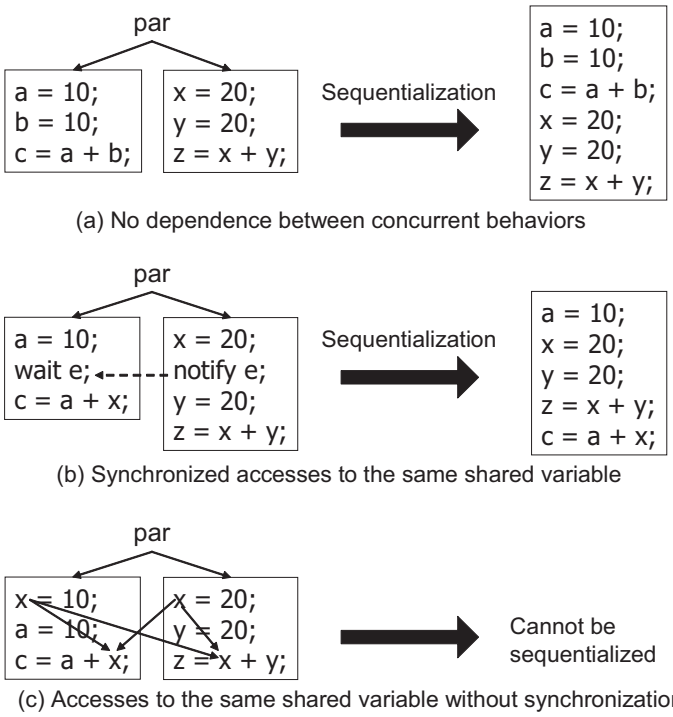


Figure 3.2: Examples of sequentialization

of hardware parts. Table 3.1 includes the numbers of *behavior*(Bhvr in the table), *channel* (chnl), *par* statements, and synchronization points (Sync).

IDCT1 is an original specification of IDCT consisting of two modules for the row calculation and the column. It does not include any parallel executions. In **IDCT2**, two modules are assigned to compute each of the row calculation and the column, and they are running in parallel. In **IDCT3**, we prepared eight parallel modules for each calculation, which results in including more parallelism. **IDCT4** is obtained by explicitly inserting parallelism within each module in **IDCT3**. It means that statements that are independent from each other are identified and scheduled to execute at the same time.

VocSpec is a given specification of Vocoder (Voice decoder). It is expected to execute all on one processing element. **VocSpec** has some parallel executed functions that are independent from each other. **VocArch** is refined from **VocSpec**, by partitioning a part of the behaviors to DSP (Digital Signal Processor). **VocSched**

is a design after scheduling in each processing element.

MP3DEC1 is an MP3 decoder design in SpecC as a specification. In **MP3DEC1**, the decoder is executed by one processor and one extra hardware. In **MP3DEC2**, the behaviors to compute Discrete Cosine Transform, which is one of the most time-consuming parts, are executed by two different hardware modules in order to improve the performance.

The experimental results of the sequentialization are shown in Table 3.1. The columns indicated as # of Deadlock, # of ILP run, and Time are corresponding to the numbers of deadlock detected by the method [26], the numbers of the runs of ILP solvers to check the execution orders between parallel statements, and the total run times for the all processes of the sequentialization, respectively. Since shared variables are found in **IDCT4** and **MP3DEC2**, the property checking by the ILP solver is required to be executed. In the other cases, there is no dependence among the parallel behaviors, and they have been sequentialized only considering that the timing constraints from the synchronizations are satisfied. In the examples of MP3 decoder and Vocoder, we detected a deadlock in a channel. In those cases, we continued the experiments after modified the descriptions not to occur deadlock. In the examples of Vocoder and MP3 decoder, although the total run times take several minutes, the run times of the ILP solver are less than one second even in those large designs. We have observed that the most of the run time for the sequentialization spends in the deadlock verification.

3.5 Conclusion

In this chapter, a method to sequentialize parallel behaviors are proposed. The method can be used to check the equivalence of designs before and after parallelization and changing the schedule. By applying sequentialization, the required computation of symbolic simulation can be reduced since a large number of possible execution orders among parallel behaviors is represented by a sequentialized one. The sequntialization is realized by checking the execution order of two parallel statements in write-write or read-write dependence relation. If the order is always the same, they can be sequentialized. Through the experiments, the method can

Table 3.1: Characteristics of example designs and the experimental results

Benchmark	LOC		# of Bhvr	# of Chnl	# of *par*	# of Sync	# of Dead-lock	# of ILP run	Time (sec)
	(orig)	(seq)							
IDCT1	300	300	4	1	0	0	0	0	0.7
IDCT2	314	298	6	1	8	0	0	0	0.8
IDCT3	256	251	4	1	2	0	0	0	0.7
IDCT4	390	360	18	1	4	10	0	48	1.0
VocSpec	9165	9165	102	4	10	4	1	0	39.0
VocArch	10178	10164	144	14	15	14	1	0	48.5
VocSched	10139	10132	144	14	2	14	1	0	42.0
MP3DEC1	8580	8580	44	6	4	6	1	0	160.2
MP3DEC2	8576	8560	46	6	5	10	1	18	162.2

work on large designs in practical time.

Chapter 4

Efficient EqvClasses Generation During Symbolic Simulation Utilizing Differences between Designs

4.1 Introduction

As described in Chapter 2, equivalence checking based on symbolic simulation suffers from a large number of the executions of the procedure to generate EqvClasses (Equivalence Classes). In this chapter, to reduce the problem, we propose an efficient method utilizing the difference between two design descriptions. In the proposed method, an EqvClass is generated without calling the procedure to check the equivalence of variables when the target variables are decided to be clearly equivalent based on the difference and dependence. The method can make the verification faster when the two design descriptions have the same control structures.

4.2 Verification Flow

In this section, our proposed verification method is described. The flow of verification is shown in Figure 4.1.

As initial inputs, two designs to be verified are given as functions written in C language. The variables corresponding to input and output signals in the functions (called input variables and output variables, respectively) are defined by designers. Our method verifies whether all output variables are equivalent when all input variables are assumed to be equivalent.

After input variables and output variables are given, chopping is carried out from input variables to output variables. This operation extracts only parts of descriptions that are affected by input variables and affect output variables. Therefore, only the extracted descriptions are verified during symbolic simulation.

4.2.1 Pre-processes

First, for convenience, some pre-processes such as in-lining of macro definitions are carried out on the given descriptions. This is done by GCC pre-processor with the appropriate options. Then, the user defined functions that do not affect functionalities of designs are removed from the descriptions. For example, input/output functions such as $scanf$ and $printf$ are removed.

When there exist loop structures in the descriptions, they are unrolled. If the

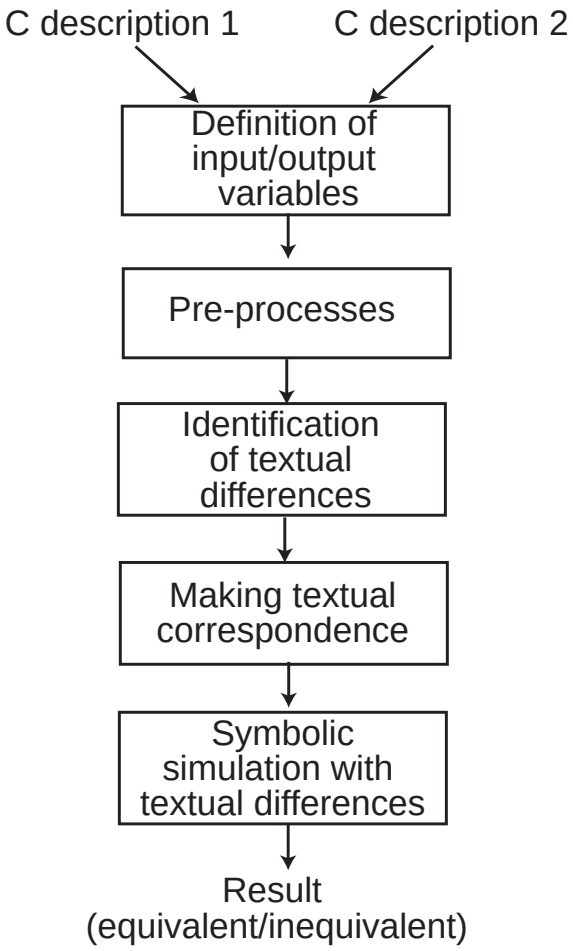


Figure 4.1: Verification flow

number of iterations of a loop is fixed, the loop is unrolled by the same times as the number of iterations. On the other hand, if the number of iterations is infinite or dependent on input variables, the number of unrolling is specified by users. If the number of unrollings is not large enough, some possible execution paths in the original descriptions are removed from the descriptions after loop unrolling. Therefore, the completeness of equivalence checking depends on the number of unrollings if the loop unrolling is carried out.

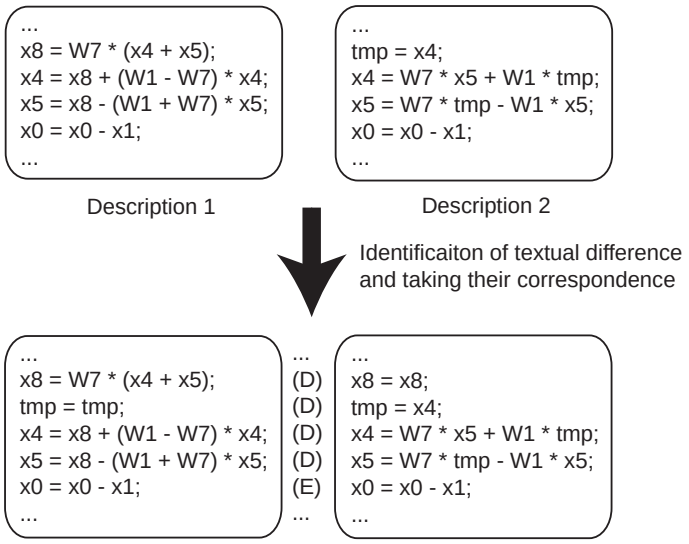


Figure 4.2: Correspondence between expressions in the descriptions

4.2.2 Identification of Differences

After pre-processes, textual differences between given descriptions are identified. This is done by using the standard UNIX command "diff". After textual differences are identified, we can take textual correspondence between descriptions. By using information of textual differences, we can establish one-to-one correspondence between expressions in the two descriptions.

Figure 4.2 shows an example of the textual correspondence between the descriptions. If the corresponding expressions are textually equivalent, they are marked as "E". On the other hand, if the corresponding expressions are textually different, they are marked as "D". Like the expression for the variable *tmp* in Description 2 of Figure 4.2, if an assignment appears on only one of the descriptions, a dummy assignment like $tmp = tmp$; is inserted in the other description to take the correspondence.

To take textual correspondence between descriptions, the proposed method receives only two descriptions that have the same control flows. In other words, we can verify equivalence of a refinement carried out on a design as long as it does not change the control flow of the design.

4.2.3 Symbolic Simulation

After the processes described above are carried out, symbolic simulation to check the equivalence of output variables is carried out. In the next sub-section, we explain symbolic simulation based on textual differences in details.

4.3 Efficient Symbolic Simulation and Generation of EqvClasses

In Chapter 2, equivalence checking in terms of symbolic simulation was introduced. To find equivalent variables, every EqvClass is checked whenever a new EqvClass is created. However, it results in that equivalence checking of variables increases with the square of the size of simulated descriptions. To reduce the number of equivalence checking of variables between the descriptions, our proposed method uses textual differences which are identified before simulation.

The flow of our proposed method to check the equivalence of a pair of expressions is shown in Figure 4.3. Depending on whether the pair is marked as "E" or "D", the way to simulate and create EqvClass is different.

If the pair is marked as "E" and not affected by variables whose equivalence is not proved, a new EqvClass for the pair is created without checking the equivalence. On the other hand, if the pair is marked as "D" or affected by variables whose equivalence is not proved, the equivalence between expressions is verified. After the verification, if they are proved as equivalent, two EqvClasses for the expressions are merged. Otherwise, our proposed method evaluates whether these expressions are for output variables or not.

If these expressions are assignments for output variables, our method terminates verification and shows all EqvClasses created during symbolic simulation as a counterexample. On the other hand, if the expressions are assignments not for output variables, successors for the pair of simulated expressions are computed by using program slicer in order to identify expressions which are directly affected by this pair. Later, when the simulation reaches the expressions identified as successors for inequivalent variables, the equivalence of variables assigned by these expressions must

be verified, because such variables are affected by the variables whose equivalence is not proved.

In our proposed method, equivalence checking of variables is omitted if pairs of expressions are textually equivalent and not affected by variables whose equivalence is not proved. Therefore, our proposed method is very efficient when two given descriptions are close to each other, because the equivalence checking between variables is applied only few times. As a result, we can save the verification time significantly.

Example

We explain our proposed method with a simple example shown in Figure 4.4. Initially, the input variables $in1$ and $in2$ are assumed to be equivalent in both descriptions. We verify whether the output variable *out* is equivalent or not in both descriptions. Note that after textual correspondence is taken, all variables in description 1 are denoted as v_1, whereas all variables in description 2 are denoted as v_2.

In the first "D", two EqvClasses for a_1 and a_2 are created. Then, the equivalence of a_1 and a_2 are verified. Since they are not equivalent, successors for a_1 and a_2 are computed to identify expressions which are directly affected by a_1 and a_2. The assignments to the variable e_1 is identified in description 1, whereas the assignments for the variable e_2 is identified in description 2.

In the first, second, and third "E", three EqvClasses are created without checking the equivalence of b_1 and b_2, c_1 and c_2, and d_1 and d_2. This is because corresponding expressions are textually equivalent and they are not affected by variables whose equivalence is not proved.

In the fourth "E", two EqvClasses for the variable e_1 and e_2 are created separately, although they are marked as "E". This is because these variables are affected by not equivalent variables a_1 and a_2. Then, we can identify that the variables e_1 and e_2 are not equivalent by equivalence checking. Therefore, successors for e_1 and e_2 are computed. As a result, the assignments to the variables out_1 and out_2 are identified.

Finally, in the last "E", two EqvClasses for variables out_1 and out_2 are created.

Since they are not equivalent because of the effect from e_1 and e_2, we can conclude that the output variable *out* is not equivalent between descriptions.

4.4 Experimental Results

A prototype tool of our proposed method has been implemented in C language. In the tool, the basic idea of equivalence checking by symbolic simulation is taken from the method in [24]. CodeSurfer [37] is called when slicing of a description is required, while CVC [40] is called when the equivalence of variables cannot be verified by symbolic simulator. All other parts in the tool were newly developed. The experiment was carried out on the PC with Xeon 2.4 GHz processor and 2GB memory.

For experiments, we prepared two example descriptions in C language. One is Inverse Discrete Cosine Transformation (IDCT) from MPEG2 [42], and the other is Rijndael [43] which is one implementation of Advanced Encryption Standard (AES). Table 4.1 shows the statistics of the examples. Below, we introduce them briefly.

The original IDCT description has two functions *idct_row* and *idct_col*, both of which are 43 lines long. From the original description, we optimized it in the way shown in Figure 4.2. This optimization realizes the reduction of execution cycles. The descriptions *idct_row*1 and *idct_col*1 were correctly optimized from the original *idct_row* and *idct_row*, respectively. On the other hand, *idct_row*2, *idct_row*3, *idct_col*2, and *idct_col*3 were also changed from the original descriptions in the same way, but inserted bugs intentionally.

To carry out experiments on larger examples, we unrolled the original IDCT descriptions in eight times. This is because we can consider each unrolled function as one functional block, since the functions *idct_row* and *idct_col* are executed eight times sequentially. After unrollings, the same optimization was carried out on the unrolled descriptions and we obtained two examples *idct_row_unroll*1 and *idct_col_unroll*1 from the two unrolled functions. The other unrolled examples were inserted bugs into them.

We prepared two more examples by unrolling the encryption function of the original Rijndael descriptions. The unrolled description is 1155 lines. The example

*rijndael*1 was obtained by adding the equivalent transformations that decomposed 4-Xor operations into 2-Xor operations, while the example *rijndeal*2 was obtained by intentionally inserting bugs.

In the experiment, to show the efficiency of our proposed method, we compared the total verification time and the number of equivalence checking when our proposed method and the previous method in [25] were applied to the examples. The difference between the methods is as follows:

- **Our method** Using textual differences to reduce the number of equivalence checking during symbolic simulation as described in the previous sections. Only the corresponding pairs of assignments are checked their equivalence.

- **The method in [25]** Simulating the whole descriptions separately with equivalence checking. All variables and expressions in both descriptions are checked their equivalence.

The experimental results are shown in Table 4.2. As shown in the table, we could obtain correct results in all experiments both when applying our method and when applying the method in [25]. Also, our method could reduce the numbers of internal equivalence checkings in all experiments.

Since our method constructs dependence graph at the beginning of verification, the verification time of our method are longer than the method in [25], when the number of symbolically simulated statements were relatively small. Otherwise, our proposed method reduced the verification time, for example, by 69 % in *idct_row_unroll*1, by 25 % in *rijndael*1, respectively.

In the verification of *rijndael*2, our method took longer than the method without using textual differences. This is because almost all statements are verified their equivalences by using CVC when the descriptions are not equivalent. To avoid this problem, random simulation before formal equivalence checking seems to be effective since in most of inequivalent cases random simulation can efficiently detect the inequivalence rather than formal equivalence checking. Therefore, at first, random simulation should be operated to find out descriptions that are not equivalent to each other, then our formal method will be applied to prove the equivalence.

To show the relation between the verification time and the number of different statements, we have experimented on the unrolled *idct_row* function with incrementally adding differences. In this experiments, all added differences were equivalent. They were not only the refinements shown in Fig. 4.2 but other simple equivalent transformations. The result is shown in Table 4.3. As shown in the table, the number of internal equivalence checkings, CVC usages, and the verification time increased with the number of different assignments. Compared to $exp1$, we can see that the total verification time was dominated by the execution time of CVC. In our method, if a pair of different assignments cannot be proved to be equivalent with substitution, CVC is called to prove the equivalence. Since the differences between $exp2$ and $exp3$ were simple transformations from subtraction assignments ($a -= b;$) to subtractions ($a = a - b;$), the number of CVC calls did not change and the verification time increased slightly. From the table, we can say that our method works efficiently, especially when the descriptions have a small number of differences.

We also verified the example *idct_row_unroll*1 and *idct_col_unroll*1 by SAT-based method proposed in [8] with Bounded Model Checker for ANSI-C (CBMC) [41]. In the experiments, the C descriptions and the property that represented the equivalence of output variables were transformed into bit equations and verified by SAT-solver. Although CBMC created the SAT formula successfully, the number of the clauses were simply too many (over one million) so that the SAT formulas could not solve it within five hours. Compared to the method, our method could verify the equivalence efficiently.

4.5 Conclusion

In this chapter, we proposed an equivalence checking method for two C descriptions by means of symbolic simulation. To verify the equivalence efficiently, the method identifies textual differences between descriptions and utilizes them well so that the number of equivalence checkings is reduced. The proposed method is particularly useful when two large descriptions with few differences are given. This fact has been confirmed by the experimental results.

From the experiments shown in this chapter, the following observations are ob-

Table 4.1: Characteristics of examples

Example name	Equivalence	Description size	# of diff.	# of diff. lines
*idct_row*1	Equivalent	43 lines	3 parts	9 lines
*idct_row*2	Not equivalent	43 lines	3 parts	9 lines
*idct_row*3	Not equivalent	43 lines	4 parts	10 lines
*idct_col*1	Equivalent	43 lines	3 parts	9 lines
*idct_col*2	Not equivalent	43 lines	3 parts	9 lines
*idct_col*3	Not equivalent	43 lines	3 parts	9 lines
*idct_row_unroll*1	Equivalent	316 lines	24 parts	72 lines
*idct_row_unroll*2	Not equivalent	316 lines	24 parts	72 lines
*idct_row_unroll*3	Not equivalent	316 lines	24 parts	72 lines
*idct_col_unroll*1	Equivalent	316 lines	24 parts	80 lines
*idct_col_unroll*2	Not equivalent	316 lines	32 parts	80 lines
*idct_col_unroll*3	Not equivalent	316 lines	25 parts	73 lines
*rijndael*1	Equivalent	1155 lines	60 parts	180 lines
*rijndael*2	Not equivalent	1235 lines	90 parts	110 lines

tained.

- In the proposed method, the number of calling the equivalence checking procedure to check the equivalence of variables and expressions can be reduced.
- Both of the proposed method and the previous method, the verification take very long time when there are many execution paths in the design.
- Even if there are many paths, the verification can finish in short when the method finds a path where the equivalence cannot be proved.
- When the designs under verification are not equivalent, CVC is need to be called to check the intermediate expressions, which results in longer verification time than that by the previous method.
- The execution time of SDG construction is short even when the design is relatively large.

To solve the problem of the second item, we need to have other efficient methods that can verify the designs having many execution paths in short time. From the

Table 4.2: Experimental results

	Our proposed method			The previous method in [25]		
Example	Result	Time	# of Eqv. check	Result	Time	# of Eqv. check
*idct_row*1	Eqv	1.8 sec	4.7×10^3	Eqv	0.39 sec	8.9×10^3
*idct_row*2	Not eqv	2.6 sec	7.3×10^3	Not eqv	0.38 sec	8.9×10^3
*idct_row*3	Not eqv	2.1 sec	5.3×10^3	Not eqv	0.39 sec	8.9×10^3
*idct_col*1	Eqv	1.8 sec	1.0×10^4	Eqv	0.37 sec	1.3×10^4
*idct_col*2	Not eqv	2.3 sec	1.0×10^4	Not eqv	0.37 sec	1.3×10^4
*idct_col*3	Not eqv	2.5 sec	1.2×10^4	Not eqv	0.37 sec	1.3×10^4
*idct_row_unroll*1	Eqv	559 sec	9.6×10^6	Eqv	829 sec	3.6×10^7
*idct_row_unroll*2	Not eqv	3.8 sec	1.7×10^4	Not eqv	0.78 sec	3.0×10^4
*idct_row_unroll*3	Not eqv	12 sec	1.4×10^5	Not eqv	13 sec	3.2×10^5
*idct_col_unroll*1	Eqv	543 sec	1.8×10^7	Eqv	1592 sec	5.3×10^7
*idct_col_unroll*2	Not eqv	3.5 sec	2.5×10^4	Not eqv	0.79 sec	3.9×10^4
*idct_col_unroll*3	Not eqv	112 sec	3.4×10^6	Not eqv	256 sec	8.5×10^6
*rijndael*1	Eqv	6.0 sec	1.0×10^5	Eqv	28 sec	1.2×10^6
*rijndael*2	Not eqv	344 sec	7.2×10^5	Not eqv	59 sec	2.3×10^6

third item, the method becomes more efficient if symbolic simulation is applied from the paths that are likely to be not equivalent. A possible solution to reduce CVC runtime mentioned in the fourth problem is that not calling CVC when most of the variables are not equivalent from the beginning of the paths to be checked. However, this solution give up the accurate equivalence checking with CVC and results in that real equivalent paths are concluded to be not equivalent.

Table 4.3: Experimental results with changing the number of different assignments

	# of different lines (% of all statements)	Result	Time	# of Eqv. check	CVC usage (times)
$exp1$	0 (0 %)	Eqv	3.6 sec	6.29×10^6	0
$exp2$	24 (9 %)	Eqv	145 sec	7.87×10^6	765
$exp3$	64 (24 %)	Eqv	164 sec	8.31×10^6	765
$exp4$	80 (30 %)	Eqv	240 sec	1.01×10^7	1275
$exp5$	128 (50 %)	Eqv	527 sec	1.50×10^7	2805
$exp6$	168 (64 %)	Eqv	783 sec	1.64×10^7	4080
$exp7$	200 (76 %)	Eqv	989 sec	1.86×10^7	5100

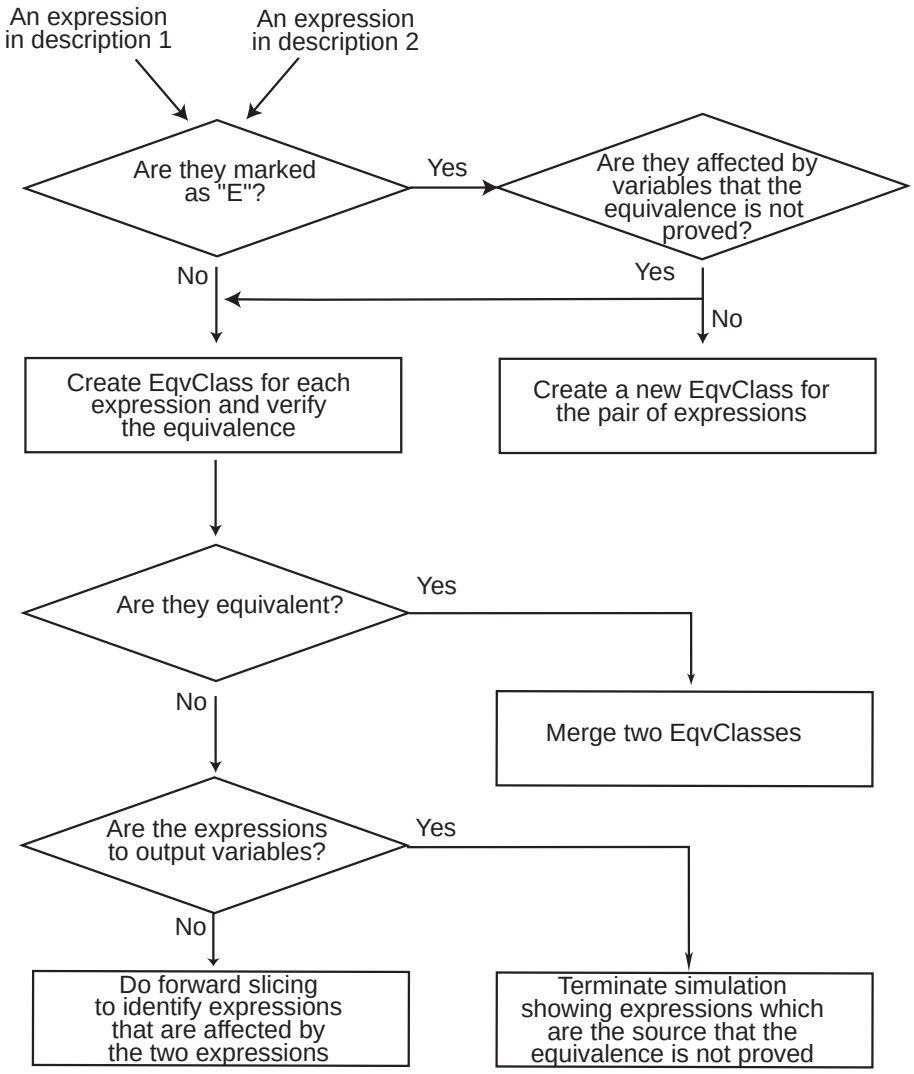
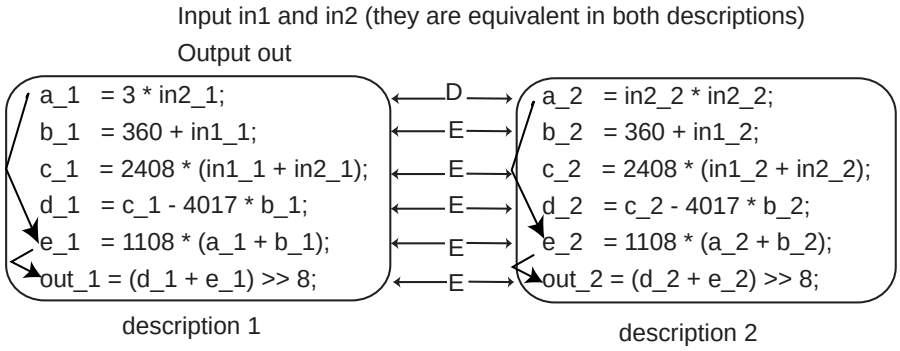


Figure 4.3: Equivalence checking for a pair of expressions



Transitions of EqvClasses

For the 1st D:
E1 = (a_1, 3 * in2_1)
E2 = (a_2, in2_2 * in2_2)

For the 1st, 2nd, and 3rd E:
E3 = (b_1, b_2, 360 + in1_1, 360 + in1_2)
E4 = (c_1, c_2, 2408 * (in1_1 + in2_1), 2408 * (in1_2 + in2_2))
E5 = (d_1, d_2, c_1 - 4017 * b_1, c_2 - 4017 * b_2)

For the 4th E:
E6 = (e_1, 1108 * (a_1 + b_1))
E7 = (e_2, 1108 * (a_2 + b_2))

For the 5th E:
E8 = (out_1, (d_1 + e_1) >> 8)
E9 = (out_2, (d_2 + e_2) >> 8)

Figure 4.4: A simple example

Chapter 5

Efficient Equivalence Checking by Applying Symbolic Simulation Locally to the Difference between Designs

5.1 Introduction

In the last chapter, an efficient method to generate EqvClasses utilizing the difference between two design descriptions. Although the method can verify faster than the previous method for the most designs, it must be applied to all execution paths in the designs, which sometimes results in the exponential increase of the verification time. As a solution of the problem, this chapter describes a verification method where symbolic simulation is applied only to the difference and its related portions. One of the most important features of the method is that it checks the equivalence of the whole designs by collecting the results of local equivalence checking for each difference. Therefore, the verification can be achieved without executing symbolic simulation for whole of the designs. On the other hand, the method extends the area of the local verification until the equivalence of the area is proved, which causes that the area becomes too large when a target difference is not equivalent. However, we can avoid this problem when we target on some specific optimizations/refinements and define the limitation of the extension of the local verification area in advance.

5.2 Proposed Method

5.2.1 Overall Flow

The overall flow of our proposed equivalence checking method is shown in Figure 5.1. As inputs, two C-based design descriptions are given with the definition of input and output variables. In addition, the correspondence of those variables between programs is given. Then, our method verifies the equivalence of the output variables by using symbolic simulation and reports the verification result ("equivalent" or "not equivalent").

In the following sections, the verification method is described in detail.

5.2.2 Restrictions of Input Design Descriptions

Our method can verify C programs that satisfy the following restrictions.

- No pointer uses (or all pointer uses are analyzed and replaced by certain variables) or dynamic memory allocation

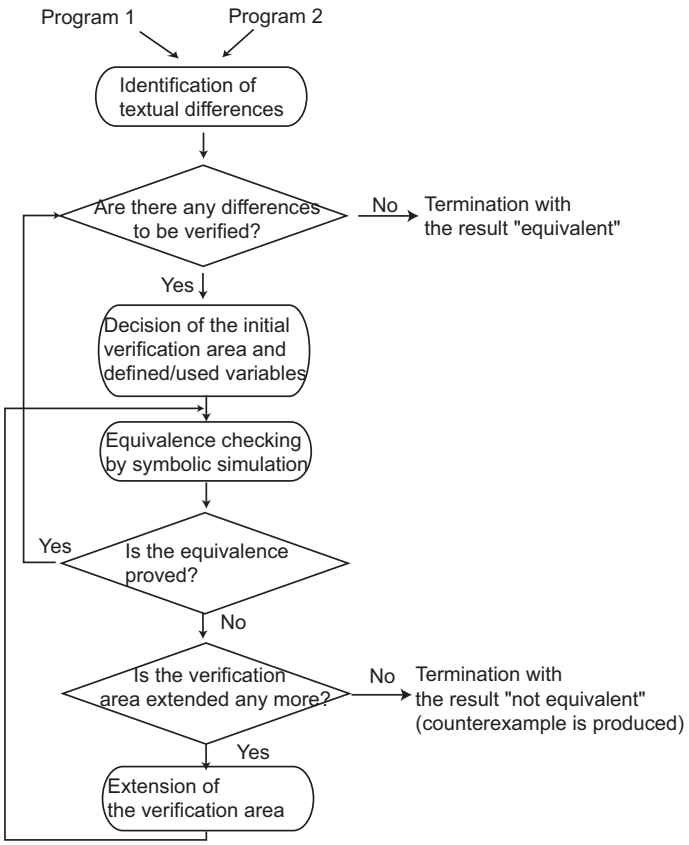


Figure 5.1: Our proposed verification flow

- Loops are unrolled in a certain times
- No recursive function calls

These restrictions come from the limitation of symbolic simulation. Therefore, if these statements are out of the verification by symbolic simulation, it does not matter whether given C-based design descriptions have these statements or not. Currently, our symbolic simulator aborts the verification when these statements are appeared during symbolic simulation. In practice, however, refinements that are not related to these statements are often carried out. In such cases, our symbolic simulator can carry out verification.

5.2.3 Identification of Textual Differences

In our method, textual differences are identified at first by using UNIX command "diff" to get hints where the equivalence of programs must be verified.

For the purpose of making correspondence between statements in both descriptions, dummy statements are inserted to the descriptions in the following cases.

- When an assignment is removed, the assignment to the same variable such as $a = a$; is inserted.
- When a conditional branch is removed, the same branch structure is inserted where all assignments are replaced by ones to the same variable.

Since these inserted statements clearly preserve the original behavior, the result of verification is not changed. Even if many statements are different, the descriptions after inserted the dummy statements cannot be twice as large as the original descriptions.

Then, SDGs for both descriptions are constructed. At the same time, statements are removed from SDGs when they do not affect any output variables and are not affected by any input variables. This reduction can be performed on SDGs and effective when users specify intermediate variables as inputs/outputs.

5.2.4 Initial Verification

In our method, a verification area can be presented by a set of SDG nodes since each node corresponds to a statement in C descriptions. The initial verification area for a difference is two sets of SDG nodes corresponding to the difference (one set from each description). Note that a difference may consist of several statements. We define input variables and output variables of a local verification area as below.

- **Local input variable** a variable corresponding to a data dependence edge coming from out of the verification area to the area
- **Local output variable** a variable corresponding to a data dependence edge coming from the verification area to out of the area

Only when a variable is a local output variable in each description, its equivalence is checked in the verification. Although other local output variables are not checked for this difference, they will be taken into account in verification for other differences if required.

A pair of corresponding local input variables is equivalent in the following cases.

- They are not affected by any differences that are prove to be inequivalent.
- They are already proved to be equivalent by the verification of another difference.

In the verification, equivalences of other pairs of local input variables are considered to be unknown variables.

If all pairs of local output variables are proved to be equivalent, the verification area of the difference is also proved to be equivalent. On the other hand, if the equivalence of any local output variables are not proved, the verification area is extended so that preceding and/or succeeding statements are included.

5.2.5 Extension of Verification Area

If the equivalence checking for a local verification area is not proved, the area is extended based on the dependence relation. The reason why the extension is required is that the equivalence of a difference can be proved after extending the verification area.

We define some extensions of the verification area as below.

- **Backward extension** Adding a directly preceding SDG node that has a data dependence to any local input variable
- **Forward extension along data dependence** Adding a directly succeeding SDG node that has a data dependence from any local output variable
- **Forward extension along control dependence** Adding all directly succeeding SDG nodes that have a control dependence from any local output variables (This extension can be carried out if any condition nodes are proved to be inequivalent)

In extension, multiple SDGs that presents assignments to the same variable are added to the verification area when control dependences of them are different. In the cases, the nodes that control these assignments are also added. After the extensions, the local input/output variables are derived for the new verification area, and verification is carried out.

We also define the limitation of extensions.

- If the equivalences of added SDG nodes are already proved, no backward extension is applied from them
- If added statements are the top (or end) of programs, no backward extension (or forward extension) is applied from them

Backward Extension Adding preceding statements based on backward extension improves the possibility that the verification is proved. This is because additional information for variables is used for the verification of the initial defined variable. We explain this effect using an example.

Figure 5.2 shows an example that the equivalence of a difference (the last statements in both programs) is proved by using backward extension. At first, the equivalence checking for the defined variable out_0 in the difference (A1) is not proved because the equivalence of the used variables (i.e., a_1, b_0, and c_0) is not proved. By applying backward extension one time from all used variables in the difference, the verification area is extended to A2.

In the verification area A2, the equivalence of defined variables a_1, c_0, d_0, and out_0 is verified in this order. However, because of the existence of unknown variables a_0 and b_0, the equivalence of all these defined variables is not proved. Therefore, the verification area is extended to A3 by applying backward extension again from all used variables in A2.

In the verification area A3, the equivalence of defined variables a_0, b_0, a_1, c_0, d_0, and out_0 is verified in this order. From the verification, we can identify that the initial defined variable out_0 is proved as equivalent.

Forward Extension Adding succeeding statements based on forward extension avoids the possibility of "false negative". There is a possibility that a difference

related to output variables is not proved or proved as not equivalent. However, output variables may be equivalent because of the effect from other statements. Therefore, if we conclude that output variables are not equivalent when a difference is not proved or proved as not equivalent, false negative is happened. We explain how forward extension avoids false negative by using an example.

Figure 5.3 shows another example that the equivalence of a difference (statements before the last in both programs) is proved as not equivalent but the equivalence of the output variable *out* is proved by using forward extension and backward extension.

At first, the equivalence of the defined variable a_1 in the difference (A1) is not proved because the equivalence of the used variables (i.e., b_0 and d_0) is unknown. Therefore, we apply forward extension from the defined variable a_1. The verification area is changed to A2.

In the second verification, the defined variables a_1 and out_0 are simulated in this order. In this case, the equivalence of a_1 and out_0 is not proved because the equivalence of the used variables (b_0, d_0, a_1, and c_0) is unknown. Because the end of programs is reached by the forward extension, we apply backward extension from all of the used variables in the verification area A2. This results in that the verification area is extended to A3.

In the third verification, the defined variables b_0, c_0, a_1, and out_0 are verified in this order. From the verification we can conclude that the output variable *out* is equivalent in both programs.

5.2.6 Symbolic Simulation on SDGs

In this work, symbolic simulation presented in section 3 is used at SDG-level. To preserve dependence relations, if a data/control dependence from a node A to a node B exists, A must be symbolically simulated before B is simulated. The ordering can be realized by topologically sorting all SDG nodes in the verification area.

5.2.7 Verification Example

We show how our proposed method works on an example shown in Figure 5.4. We assume that the variables $in1$ and $in2$ are the primary inputs of the program, and

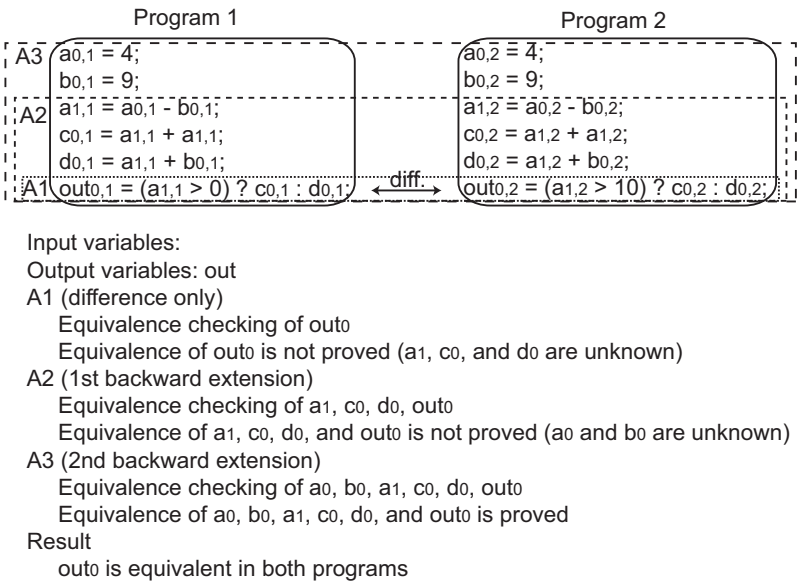


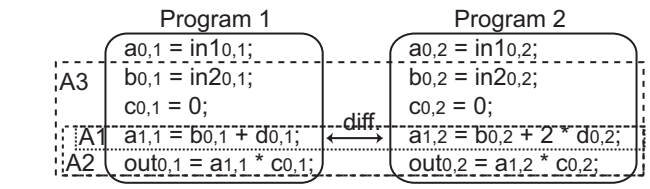
Figure 5.2: Equivalence checking with backward extension

the variable *out* is the primary output. The statement $x = x$; in Description 1 is added as a dummy statement to make a correspondence to $x = x + c$; in Description 2.

At first, the first difference $D1$ is verified. The first verification area is A in the figure, and its local input variables are a and c, and its local output variable the variable x. Since all local input variables are unknown, the equivalence of x cannot be proved. Thus, in this case, we decide to backwardly extend the area from a.

Then, the extended verification area become the area B, and the verification is carried out again. In this case, the local input variables are $in1$, $in2$, and c, and the local output variables are x and $(in1 > in2)$. Since the equivalence of x cannot be proved after the verification with the area B, we decide to forwardly extend the area from x and obtain the area C.

After the verification with this area C, we can prove the equivalence of x. The verification for the difference $D2$ is not carried out, since it is included the verification for $D1$. Then, as the all difference is verified, it can be said that the two descriptions are functionally equivalent.



Input variables: in1, in2
Output variable: out
A1 (difference only)
  Equivalence checking of a_1
  Equivalence of a_1 is not proved (b_0 and d_0 are unknown)
A2 (1st forward extension)
  Equivalence checking of a_1 and out_0
  Equivalence of a_1 and out_0 is not proved (b_0, d_0, a_1, and c_0 are unknown)
A3 (1st backward extension)
  Equivalence checking of b_0, c_0, a_1, and out_0
  Equivalence of b_0, c_0 and out_0 is proved
Result
  out_0 is equivalent in both programs

Figure 5.3: Equivalence checking with forward extension

5.2.8 Discussion on Strategy of Extension

In general, a verification area can have multiple local input/output variables. Therefore, there are a number of combinations to apply backward and forward extensions. This makes us difficult to define the best strategy of extensions. In the followings, we list some reasonable strategies for differences usually happen in practice.

- Applying backward extensions until the start points of the programs, then applying forward extensions until end points
- Applying forward extensions and backward extensions in turn
- First, applying backward extensions m times, then applying forward extensions n times (m, n are pre-defined number)

These strategies are similar to ones in equivalence checking of gate-level circuits.

In some cases, designers know which kinds of refinements are carried out. In such cases, a specific strategy for the refinement can be applied to improve the verification speed.

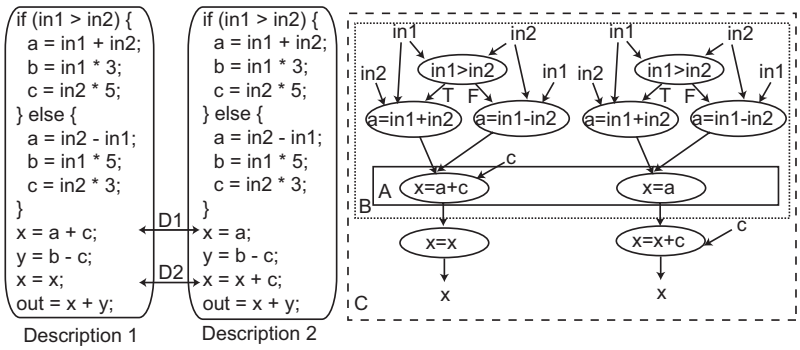


Figure 5.4: Equivalence checking example

Application 1: Common Subexpression Elimination Common subexpression elimination is executed to reduce the number of operations. Figure 5.5(a) shows an example of a common subexpression elimination. As textual differences, two kinds of statements are identified. One is the newly added statement to assign the common subexpression to a temporary variable (The statement for the define variable t_0 in the program 2). The other is the set of differences where the temporary variable is used (The statements for the defined variable x_0 and z_0 in both programs).

The transformation is validated efficiently by applying our method. We assume that used variables a_0, b_0, c_0, and d_0 are equivalent in both programs.

First, the difference where the variable x_0 is defined is verified. Note that the difference for the variable t_0 in the program 2 is not verified because the correspondence of the variable between programs is not taken (i.e., t_0 does not appear in the program 1). In this case, the equivalence is not proved because of the existence of an unknown used variable t_0 in the program 2. Therefore, we apply backward extension one time for the used variable t_0 (The verification area is extended to A2). Next, the equivalence of x_0 is verified again by using the information of the used variable t_0. In this case, the equivalence is proved as equivalent.

The equivalence of the defined variable z_0 is verified without applying any extension because the information for the used variable t_0 is already known.

From the result, only one extension is enough to verify the equivalence of programs for this transformation.

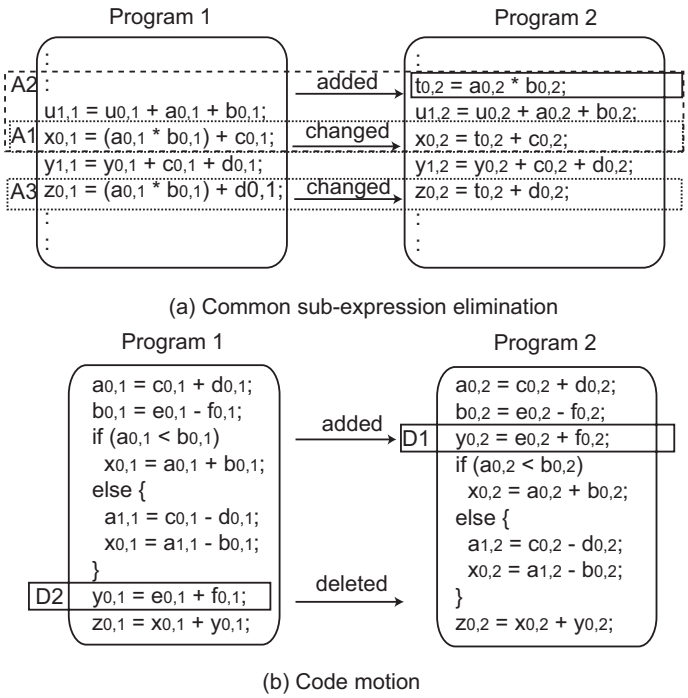


Figure 5.5: Examples of source-to-source transformations and optimizations

Application 2: Code Motion There are many kinds of code motion for several purposes. In this case, we consider a simple code motion where assignments are moved along a single path without across any conditional branches. Figure 5.5(b) shows an example of this transformation. In this example, we assume that the used variables e_0 and f_0 are equivalent in both programs.

As the textual differences, two differences (D1 and D2) are identified. Looking through these statements, we can take the correspondence because they have the same defined variable y_0 and the same used variables e_0 and f_0. Then, we verify the equivalence of the defined variable y_0 in both programs. In this case, we can verify the equivalence without applying any extensions.

5.3 Experimental Results

We have implemented the proposed method with CodeSurfer[37] and CVC[40]. CodeSurfer is used to construct SDGs of programs to be verified. The experiments

were performed on the following design examples written in C language.

- Common sub-expression eliminations in a differential equation solver (total 130 lines, differences are 10 parts, 30 lines)
- Refinements in IDCT(Inverse Discrete Cosine Transform) (total 420 lines, differences are 16 parts, 96 lines) from MPEG2 program[42]
- Refinements from 4-Xor into 2-Xor in the encryption function (total 1235 lines, differences are 40 parts, 120 lines) from Rijndael program[43]

The refinements in IDCT is to reduce the computation, and it has applied combinations of common sub-expression elimination and factorization. All experiments were carried out on PC with 2.4 GHz processor and 2 GB memory.

The experimental results are shown in Table 5.1. All verification results are same as what we have intended. As shown in the table, the numbers of SDG nodes that are symbolically simulated are much smaller than the total SDG nodes in the programs. This is seen especially in the inequivalent cases. This is because the result can be concluded to be inequivalent if a counterexample is found.

As for the comparison of verification times with the method that symbolically simulates the whole programs, the proposed method takes shorter times to verify when the verified programs are relatively large. For example, equivalence checking with symbolic simulation of the whole IDCT example, which has eight conditional branches, takes more than 800 sec, while the proposed method takes 1.8 sec as shown in the table. On the other hand, diffeq and rijndael examples can be solved within 1 sec by both of the two methods.

In addition, symbolic simulation for the whole MPEG2 or Rijndael cannot be carried out in practical time. Therefore, our approach where only the portions related to the differences are symbolically simulated is effective especially when a given program is very large.

5.4 Conclusion

In this chapter, an equivalence checking method that applies symbolic simulation to local verification areas based on the difference. The method can verify faster

Table 5.1: Experimental results

	result	time	verified nodes	total nodes
diffeq1	eqv	0.7 sec	60	288
diffeq2	ineqv	0.7 sec	73	288
mpeg1	eqv	1.8 sec	192	1160
mpeg2	ineqv	0.9 sec	62	1160
rijndael1	eqv	0.3 sec	240	4112
rijndael2	ineqv	0.6 sec	44	4112

compared the method that applies symbolic simulation to whole of the designs. The proposed method extends the verification areas that are begin checked repeatedly along dependency, until the equivalence of the area is proved. This extension is required to avoid reporting the result "not equivalent" when a difference is locally not equivalent but whole of the designs are equivalent. Therefore, if the designs have small equivalent differences, the verification speed is improved significantly. In the method, it is important to decide to extend the local verification areas for which variables in which direction (forward or backward). Considering a specific optimization/transformation, the strategy of the extensions can be properly decided.

Chapter 6

Equivalence Checking for Loop Optimization without Unrolling

6.1 Introduction

Both in hardware and software, loop executions are sometimes critical to the total performance of the system, which results in the great demands of the optimization. To optimize loops, designers usually need to refine the source code manually, since compilers or behavioral synthesizers used in practical cannot apply state-of-the-art optimization techniques. Therefore, the equivalence checking of such optimizations is required to verify the manual optimizations.

The main purposes of optimizing loops can be summarized as follows:

- To remove dependences among different iterations of loop executions (which enables compilers to find more candidates of optimization and introduce more parallelism)
- To reduce the array accesses (which results in the reduction of time-consuming memory accesses in both hardware and software)

We can see that loop optimization plays an important role in hardware designs, since reducing the number of memory accesses and introducing more parallel executions can significantly improve the performance of the hardware products. Loop optimization in source codes is also required to help compilers to find more candidates of further optimizations such as code motion and speculative execution.

In this work, an equivalence checking method for such loop optimizations is proposed. It is a formal verification technique, hence, it needs no input test pattern for verification and checks equivalence by mathematical reasoning. The target of loop optimization in this work is optimizations for array-intensive loops. Such loops are found in designs of filters, arithmetic transforms as cosine transform, and other data computation of a sequence of variables (i.e. arrays). First, the proposed method constructs two data-flow graphs for two codes before/after optimized. Then, on the graphs, symbolic simulation based equivalence checking is applied Similar to that software programs are written in programming languages such as C language, C-based hardware description languages are proposed. They enable to describe hardware designs in more abstract level than register-transfer-level that is currently the beginning stage for most designs. This abstract design level is called system-level and

has become applied to real products. Generally, system-level designs include both hardware and software. This enables fast simulation and flexible hardware/software partitioning. Finally, the hardware parts are synthesized into RTL designs by behavioral synthesis, while the software parts are compiled to assembly codes.

One of the contributions of the work is that our verification method can accept the codes with less restrictions than those in the previous work where only arrays are basically allowed in programs under verification. On the other hand, our method can verify the codes with temporary variables, data-dependent conditional branches, and statements outside loops. Another is that our method can verify the equivalence with complicated expressions, using decision procedure with symbolic simulation. In the previous work, the equivalence that can be proved is limited to expressions having the same data-flow. without unrolling the loops, and the result will be obtained.

6.2 Proposed Verification Method

First of all, we define two terms *iterator* and *index* used in this chapter.

- *Iterator* of a loop is the loop variable that is incremented by a constant number in the loop. The executions of loops are determined when the iterator is given. In Figure 6.2, each loop has an iterator i.
- *Index* in an array access $a[i]$ is i

6.2.1 Overall Flow

The overall flow of the proposed verification method is shown in Figure 6.1. The inputs of the verification are:

- Two program codes including loops and array accesses
- Input variables and arrays that are assumed to be equivalent in both programs under verification
- Output variables and arrays to be checked for their equivalence

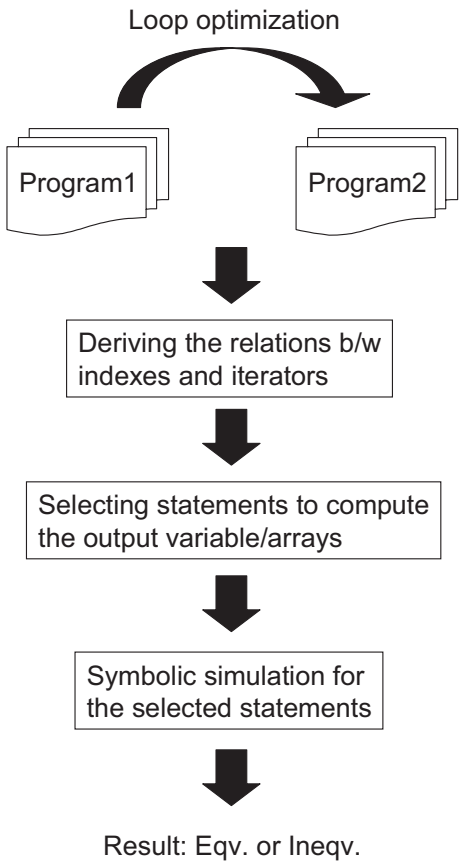


Figure 6.1: The overall verification flow

Note that the input/output arrays are specified with the ranges of the indexes. Given the inputs described above, the verification method tells whether the output variables and arrays are equivalent or not.

First, for the given two programs including loops and arrays, the relations between the iterators of the loops and the indexes where the arrays are assigned are identified. Then, based on the relations, the statements that are required to compute the output variables and arrays are selected from the original programs. At the same time, the symbolic values of the loop iterators are decided. Finally, symbolic simulation is carried out only for selected statements with the specified symbolic values of the iterators, and the equivalence is decided.

In this thesis, two arrays are said to be equivalent if and only if they have the same ranges and the all elements for the same indexes are equivalent. Therefore, we do not consider two arrays a and b as being equivalent in the following cases.

- The ranges of a and b are different. For example, arrays a and b are not equivalent when the range of a is greater than that of b, even if $a[i] = b[i]$ for each i within the range of b.
- All elements are equivalent under some index mappings between two arrays. For example, arrays a and b are not equivalent even if $a[i]$ and $b[max - i]$ are equivalent for $0 \leq i \leq max$.

6.2.2 Restrictions on Programs

In this section, we describe the restrictions on programs, especially on loops and arrays, in order to be verified by the proposed method.

- Array indexes must be affine to the iterator
- No pointer references

Compared to the method in [31], we can verify the programs with loops including non-static control-flow and multiple assignments to the same variables and arrays. The former allows to describe short-cut paths and procedures for rounding up or down, while the latter enables to use temporary variables in the loops.

6.2.3 Establishing Index-Iterator Relation

First, for each loop in the two programs, the bound and the increment step are derived. It can be represented as a tuple shown below.

$$ITER_{L_n} = (lower\_bound, upper\_bound, step)$$

Here, L_n denotes the label to identify the loops. *lower_bound*, *upper_bound*, and *step* denote the initial value, the final value, and the increment step of the iterator of a loop L_n.

For each read or write array access, the relation between the accessed index and the iterator of a loop, where the array access is located, is identified. If an accessed index is expressed directly by constants and/or the iterator, the expression itself becomes the relation. Otherwise, if an expression of an index includes variables other than the iterator, it must be transformed to an expression only using the iterator and constants, by analyzing dependency.

The index-iterator relations are represented in the following form.

$$IDX_{A,S_n} = p * ITER_{L_n} + q$$

where A and S correspond to an array accessed and the statement where A is accessed, respectively, and n denotes the number of array accesses in S in the order of appearing. Also, p and q are integer numbers.

Figure 6.2 shows a verification example. The two loops in the program shown in left-hand side in Figure 6.2 (a) is merged to the single loop in the right-hand side. The bounds and increment step of the iterators and the index-iterator relations are shown in Figure 6.2 (b). In this example, the relations can be identified only considering the indexes of the array accesses in the loops, since the all indexes are expressed directly by the iterators.

6.2.4 Constructing Data-flow Graph with Iterator Relations

The heart of the proposed method is applying symbolic simulation to the loops with specified iterators and getting the symbolic expression of the arrays of arbitrary indexes. For example, when the output array is a, the symbolic expression of $a[i]$ for arbitrary i is generated. The iterator values for symbolic simulation is determined by analyzing the index-iterator relations described above. Generally, different index i results in different symbolic expression of the array $a[i]$, since the execution paths to compute $a[i]$ can be different by i due to conditional branches. However, usually, the numbers of different symbolic expressions of the output variables and arrays are small.

Using the example shown in Figure 6.2, we briefly explain the basic idea how our method proves the equivalence of arrays without loop unrolling. The input of the verification is three arrays a, b and c, which means the arrays a, b and c in both

programs have the same number of elements and each of them are equivalent. The output is an array c. First, we have to identify the symbolic values of the iterators to assign $c[i]$ at $s4$ in both programs. Based on the relations between the indexes and the iterators, we can identify that ith iteration of $L2$ and $(i+100)$th iteration of $L1$ should be executed in Program 1 and Program 2, respectively, in order to compute $c[i]$. Similarly, we can identify what number of the iterations should be executed for $b[i-100], b[i], b[i+100]$ in $s4$ in Program 1, and $b[i-200], b[i-100], b[i]$ in $s4$ in Program 2. Repeating the same procedure, finally, the symbolic expression of $c[i]$ can be expressed only by the input array a and b.

In the followings, we give the detailed procedures to select statements and specify the iterator values to which symbolic simulation is applied.

For each output variable or array, all possible execution paths are generated with the specified iterators. It is carried out by backwardly tracing the data-flow from each output variable or array until reaching to the input variables and arrays. At that time, if the output is an array, we give a symbolic value i as the index of the array, which means that it represents arbitrary indexes of the array. Therefore, when symbolic simulation of all possible execution paths for the output array with the index i results in being equivalent, we can say that the output array is equivalent for arbitrary indexes.

The execution paths can be represented as a data-flow graph where each node represents an assignment and each edge represents a data-flow. In addition, each edge is labeled by a relation of iterators between two consecutive nodes. The data-flow graph with iterator relations has the following characteristics.

- Each edge is labeled by $n : (f(i), R_s) \rightarrow (g(i), R_d)$. n denotes the nth argument of the assignment corresponding to the source node. $f(i)$ and $g(i)$ denote index-iterator relations of the source and destination nodes, respectively. R_s and R_d denote the ranges of i at the source and the destination nodes, respectively.

- A node has a guard condition if the corresponding assignment is guarded by some conditions.

- Nodes can be merged into a single node if the following two conditions are

satisfied

- The assigned variables are the same or the same indexes of the same array are assigned
- The assignment is executed exclusively

An example of the graph is shown in Figure 6.2 (c). In the figure, A_in and A_out indicate the input and the output array A, respectively. As the node $(s1, s2)$, if the same index is assigned in multiple assignments under conditional branching, they are merged to a single node, as described above. In the figure, the ranges of the iterators are not shown to improve the readability.

6.2.5 Symbolic Simulation based Equivalence Checking

Before symbolic simulation, for each assignment node in a loop, symbolic values of the iterators must be decided in the following way.

1. Replace i in $f(i)$ and R_s of all edges going from the output arrays with I
2. Repeat the following two procedures until each node in the graph is labeled by a function $C(I)$, for all paths from the output to the node. Therefore, a node can be labeled by $C(I)$ multiple times if there are multiple paths from the output to the node.
   - For each edge where either $f(i)$ is already expressed by I, solve $f(I) = g(i)$ for i, and let the result be $C(I)$
   - Label the destination node N_d of the edge by $C(I)$
   - Replace i in $f(i)$ and R_s with $C(I)$ for all edges going from N_d
   - Solve R_s for I and obtain the range in terms of I

Sometimes, the resulting $C(I)$ at nodes may be expressed using fraction numbers. In such cases, $C(I)$ of all nodes in both of the programs are multiplied by an appropriate integer number so that fraction numbers in $C(I)$ can be removed.

The generated data-flow graphs with iterator relations, symbolic simulation is carried out in the topological order. At this point, all nodes in loops are labeled

by the function $C(I)$ for each path from the output nodes. Symbolic simulation is performed from the input to the output with setting the iterators to $C(I)$. At the same time, the ranges of the arrays are computed as the intersections of the ranges symbolically simulated so far.

During symbolic simulation, two nodes are decided to be equivalent when the following conditions are satisfied.

- The corresponding symbolic expressions of the nodes are the equivalent
- If the nodes represent arrays, the assigned indexes are the same
- The ranges of the iterators are the same

After symbolic simulation, the equivalence of the output variable or array can be decided. When checking an array A, only the symbolic expressions of $A[i]$ in both programs are compared. In the case of $C(I)$ is multiplied by an integer n, the symbolic expressions of $A[n \times i]$ are compared instead.

Example In the example in Figure 6.2 (c), every iterator function is an formula expressed by i, $i + const$, or $i - const$. Therefore, the resulting $C(I)$ labeled to the nodes become also the same forms, and need not to be multiplied by the least common multiple of the coefficients.

First, all i in $f(i)$ of all edges from c_{out} are replaced by I. Then, by solving $I = i$, we can obtain $C(I) = I$ and label the node $s4$ by I in Program 1. This means that the iterator of $L2$ should be set to I to obtain the computation of $c[I]$. At the same time, for all edges from $s3$ to $s4$, i in $f(i)$ is replaced by the resulting $C(I)$, i.e. I. Again, solving $f(I) = g(i)$ for all three edges. For example, for the edge of the first argument of $s4$, we can get $C(I) = I - 100$ by solving $I - 100 = i$ and label $s3$ with $I - 100$. This means that the iterator of $L1$ should be set to $I - 100$ when $s3$ is symbolically simulated. Note that this label is valid only for the path along the edge for the first argument.

6.3 Experimental Results

We evaluated the effectiveness of our proposed method through the experiments on several programs between loop optimizations. In the experiments, the data-flow graphs with index-iterator relations are manually generated, and they are handed to our symbolic simulator. Then, the symbolic simulator checks the equivalence of the output arrays. The symbolic simulator is implemented in C language and works with CVC, which is available from [40], to improve the ability of checking equivalence. The symbolic simulator works on PC with 2 GHz processor and 1 GB memory.

6.3.1 Examples

The following three sets of examples have been prepared for the experiments.

- Four variations of a filter program ($filter1-4$)
- Three variations of DWT (Discrete Wavelet Transform) program ($dwt1-3$)

The detailed characteristics of those example programs are shown in Table 6.1. The original programs are $filter1$ and $dwt1$, and they have two and five loops, respectively. Applying loop fusion, the number of loops in both programs are reduced to one and two, respectively. The examples have temporary variables and assignments outside loops that cannot be handled in the previous work in [31]. Also, we have prepared the programs with bugs. They are shown as $(program\_name)_{bug}$ in Table 6.2.

6.3.2 Results

The experimental results of symbolic simulation on the data-flow graphs are shown in Table 6.2. All verification results are the same as what we have expected. The execution time for analyzing data dependence graphs to determine the iterator values for symbolic simulation is within 0.1 second in each example. This is partly because the numbers of the execution paths are not very large (i.e. less than 30) in the experimented designs. Since the typical size of loops that are applied optimizations

Table 6.1: Characteristics of the example programs

	# of lines	Applied optimization
$filter1$	16	Original program
$filter2$	19	Loop fusion
$filter3$	23	Loop fusion, scalar replacement, and array duplication
$filter4$	22	Loop fusion and scalar replacement
$dwt1$	25	Original program
$dwt2$	28	Loop fusion
$dwt3$	38	Loop fusion and scalar replacement

Table 6.2: Verification results

		Result	Time
$filter1$	$filter2$	Equivalent	0.47 sec
$filter2$	$filter3$	Equivalent	1.48 sec
$filter2$	$filter4$	Equivalent	< 0.1 sec
$filter1$	$filter2_{bug}$	Inequivalent	< 0.1 sec
$filter1$	$filter4_{bug}$	Inequivalent	< 0.1 sec
$dwt1$	$dwt2$	Equivalent	< 0.1 sec
$dwt2$	$dwt3$	Equivalent	< 0.1 sec
$dwt1$	$dwt2_{bug}$	Inequivalent	< 0.1 sec

is not largely different from those of the experimented loops, we can say that the proposed method can verify the equivalence in practical.

The proposed method can verify faster than the previous method in [20], since the verification cannot terminate in practical time if symbolic simulation is applied to the programs that are totally unrolled. Therefore, the proposed approach that do not unroll the loops entirely can be said to be more efficient for the optimizations.

6.4 Conclusions

In this chapter, an equivalence checking method for loop optimization is proposed. In the proposed method, the equivalence of the output arrays is checked by applying symbolic simulation. The proposed method can verify programs including

non-static control-flow and multiple assignments to a variable since the verification is performed by symbolic simulation, instead of the uninterpreted topological analysis on dependence graphs proposed in [31]. Instead of unrolling loops in the previous symbolic simulation based method in [20], our method derives the symbolic expressions for arbitrary indexes of the output arrays in both programs under verification, and compares the two expressions. The experimental results show that the proposed method can verify the equivalence in short time. This is mainly because the number of execution paths, which must be applied symbolic simulation, can be significantly reduced by avoiding loop unrolling.

When loops are required to be totally unrolled during the proposed verification process, we should introduce inductive verification. It will enable us to efficiently verify loops in which the output array is depending on all iterations of the loop. In such cases, the proposed method generates large data-flow graphs for the all iteration executions of the programs.

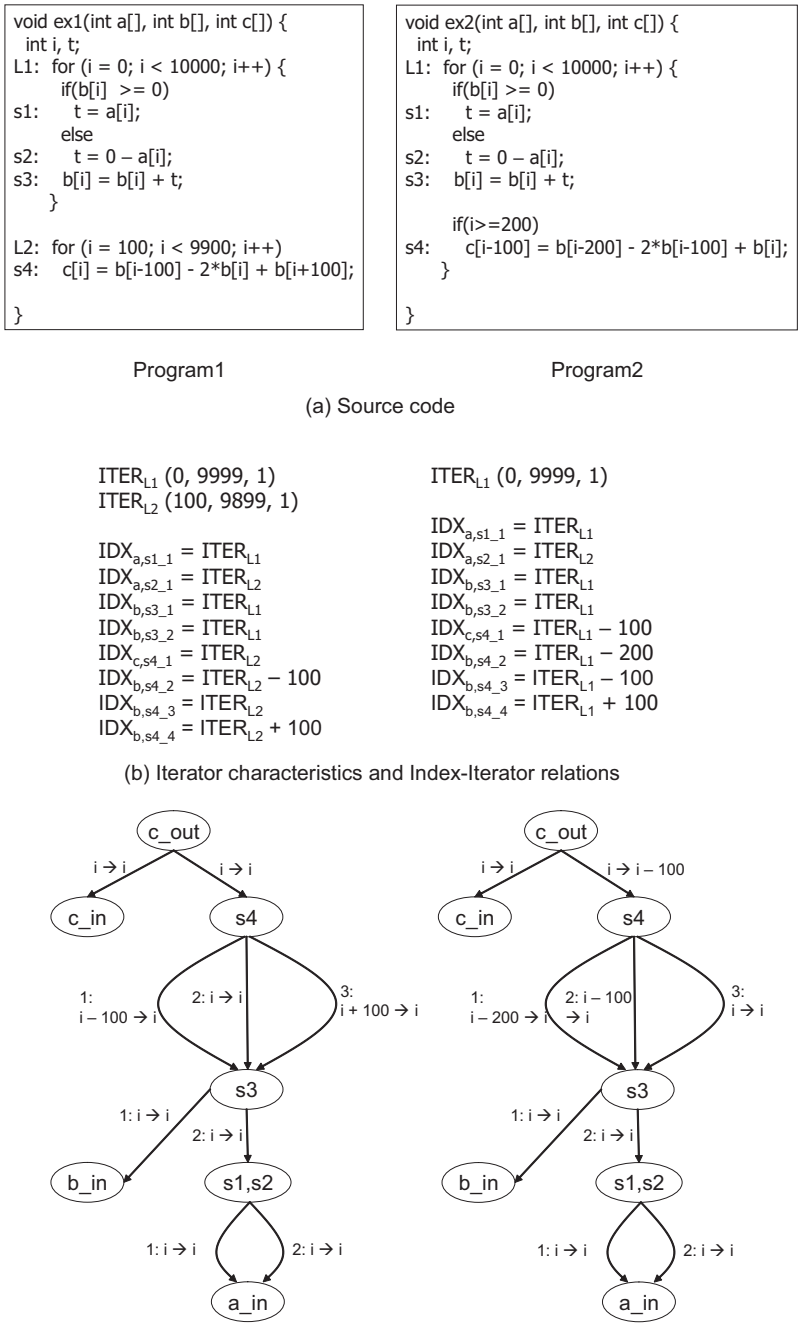


Figure 6.2: A verification example

Chapter 7

Tool Implementation and Case Study

7.1 Tool Implementation

We have developed a verification environment for system-level and RTL designs. The tool is called FLEC, which stands for Fujita Laboratory Equivalence Checker. The tool implements several formal and static methods previously presented in the papers[23, 3, 20, 19, 27, 26]. In this section, some important features of FLEC are described.

7.1.1 FLEC Overview

Figure 7.1 shows the tool architecture of FLEC. FLEC accepts two design descriptions for equivalence checking. First of all, FLEC generates ExSDGs from the given descriptions. As shown in the figure, several verification methods are implemented in FLEC, and they perform verification by reading and analyzing ExSDGs. In this sense, ExSDG can be seen as the internal representation of designs in FLEC.

To carry out equivalence checking, users have to specify the equivalence that they like to verify. This specification is given with variable correspondences, latency and throughput of variables. If the equivalence should be kept only under some conditions, they can be specified as a part of the equivalence specification. To control which verification method is applied and when it is applied, users can specify the verification methods to be applied to ExSDGs of the given designs.

Sometimes, the verification methods need to solve the validity or satisfiability of the logic formula. In such cases, reasoning tools such as CVC[40] are called and solve the problem.

The methods in double-line boxes are the implementation related to this thesis. They include the sequentialization method in Chapter 3, the symbolic simulator and equivalence management, where EqvClasses are generated and merged, in Chapter 4 and Chapter 5, and a difference extraction method.

In the following subsections, the implemented methods that are not described in this thesis are briefly introduced.

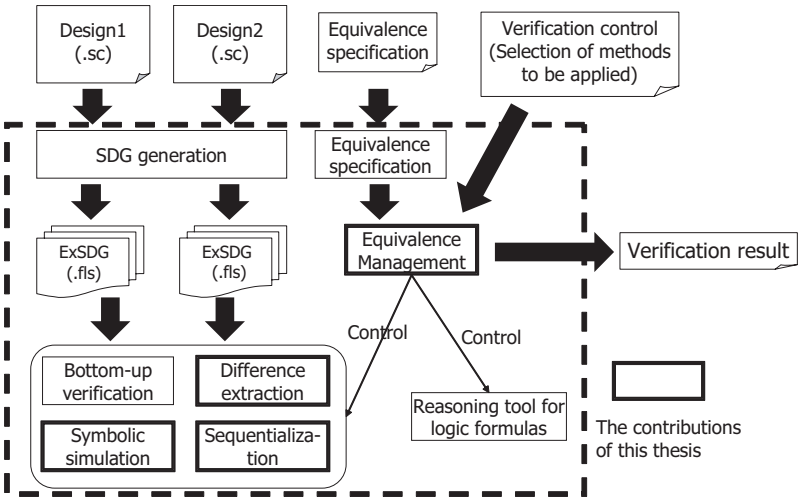


Figure 7.1: Tool architecture of FLEC

7.1.2 ExSDG Generation

Figure 7.2 shows an ExSDG generation flow from given design descriptions. To generate ExSDGs, design descriptions in C-based design language or conventional hardware design language like Verilog and VHDL are parsed and translated into internal representations. FLEC reads the internal representations and converts them into AST(Abstract Syntax Tree) defined as a part of ExSDG. Then, performing dependency analysis, ExSDGs of given designs are generated. To perform any method, since ExSDG generation is essential, the generation time should not be too long. In our experience, ExSDG generation time is usually less than one minute for design descriptions with a thousand of lines. Also, it is known that the generation time increases with the square of the design size at worst case.

7.1.3 Static Design Checking

Static design checking is a method to detect typical design errors by statically analyzing designs. In FLEC, methods proposed in [3] are implemented with ExSDG. Instead of generate and solve complex formula, the methods explore ExSDG and detect pre-defined types of graph structures as design errors. Therefore, the methods can detect all errors in designs, while they can generate false errors. Due to

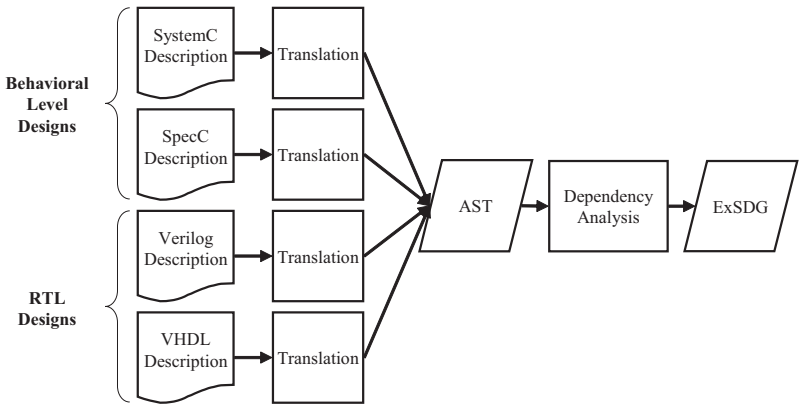


Figure 7.2: ExSDG generation flow

not solving complex formulas, the methods can work much faster than usual formal verification methods.

7.1.4 Deadlock Detection

To detect deadlocks in designs having parallel behaviors, the method proposed in [26] is implemented in FLEC. The method abstracts a given design in ExSDG and generates a set of linear equations/inequations to represent timing constraints. Then, the equations are solved by ILP(Integer Linear Programming) solvers. If a set of variable assignments that satisfies the timing condition of deadlock are found, it is checked its feasibility on a given design. If it is feasible, the method concludes that the design has deadlock. If not, the abstract model is refined[27] and checked again. The process is repeated until a feasible counterexample is found or no deadlock is found. By applying this method, we can prove that the given design is free of deadlock.

7.1.5 Difference Extraction

In the equivalence checking methods proposed in this thesis, difference between design descriptions plays an important role to improve the efficiency. To identify the difference between given two designs, AST based difference identification can be carried out. It simply compare the texts generated from AST part of ExSDG and

Table 7.1: Characteristics of MPEG4 designs in SpecC

	# of lines	ExSDG generation time (sec)	ExSDG size	
			# of nodes	# of edges
MPEG4_org	6329	778	50138	35837
MPEG4_rev1	6329	780	49754	35837
MPEG4_rev2	6329	774	49754	35837

tell users a set of nodes included in the difference.

7.2 Case Study

In FLEC, the proposed verification method presented in Chapter 5 is implemented. In this section, some case studies on real large system-level designs by the proposed verification method are shown. The method in Chapter 4 cannot be applied large designs due to the path explosion problem, as shown through the experimental results. Therefore, in the following case studies, only the method that performs local equivalence checking is applied to designs.

All of the following experiments have been done using a computer with Intel Xeon 3.0 GHz dual processors and 5.2 GB memory.

7.2.1 Case Study of MPEG4

We have prepared an SpecC design of MPEG4. In this case study, two optimizations, constant propagation and constant folding, were applied to the original design. Those optimizations can be carried out both by manually and automatically by tools like [14], and cause a great improvement when the optimized designs are synthesized. In this experiments, we have applied the optimizations to Inverse Discrete Cosine Transform (IDCT) since this function is usually implemented as hardware due to its heavy computation. The detailed characteristics of this example design are shown in Table 7.1.

The differences between designs are shown in Table 7.2. The difference between MPEG4_org and MPEG4_rev1 includes total 96 nodes from each, and the designs are equivalent since the optimization was correctly carried out. On the other hand,

Table 7.2: Difference of MPEG4 designs

Design1	Design2	# of diff	# of diff nodes in Design1	# of diff nodes in Design2	Diff time (sec)
MPEG4_org	MPEG4_rev1	48	96	96	3.8
MPEG4_org	MPEG4_rev2	48	96	96	3.9

Table 7.3: Experimental results on MPEG4 designs

Design1	Design2	Result	Time	# of ext.
MPEG4_org	MPEG4_rev1	Equivalent	3.3 sec	0
MPEG4_org	MPEG4_rev2	Inequivalent	13.2 sec	80

MPEG4_org and MPEG_rev2 are not equivalent since folded constant numbers intentionally changed to wrong numbers. In the table, "Diff time" shows the execution time to identify the difference. In this case, the times are less than 4 seconds.

The experimental results are shown in Table 7.3. In the equivalent case, the verification is done in 3 seconds. Since each corresponding difference is equivalent in the optimization, the extension of the verification areas are not needed. In the inequivalent case, the result "inequivalent" was obtained after 13 seconds run of the verification. The inequivalence is caused only by 16 differences, the remaining differences were proved to be equivalent. As a result of trying to prove the equivalence of inequivalent differences, the verification areas were extended totally 80 times (5 times for each inequivalent difference).

7.2.2 Case Study of Elevator Controller

As another case study, we have prepared an elevator controller design in SpecC. Different from the MPEG4 design, this design is control-flow intensive. To the original design, a speculative code motion was applied and Elv_rev1 was generated as a result. On the other hand, in Elv_rev2, the moved statements were changed intentionally, which resulted in Elv_rev2 was not equivalent to Elv_org. Such code motions are generally applied automatically by synthesis tools. However, ideally, the equivalence should be proved when considering the expensive cost to fix the

Table 7.4: Characteristics of the elevator controller designs in SpecC

	# of lines	ExSDG generation time (sec)	ExSDG size	
			# of nodes	# of edges
Elv_org	3349	174	20776	20825
Elv_rev1	3347	174	20774	20825
Elv_rev2	3347	175	20774	20825

Table 7.5: Difference of the elevator controller designs

Design1	Design2	# of diff	# of diff nodes in Design1	# of diff nodes in Design2	Diff time (sec)
Elv_org	Elv_rev1	3	4	4	1.9
Elv_org	Elv_rev2	3	3	3	1.9

bugs after manufacturing hardware. The characteristics of the designs are shown in Table 7.4.

The difference between designs is limited to a few nodes, and it can be identified within 2 seconds by our difference identification program. The detailed information of the difference can be seen in Table 7.5.

The results are shown in Table 7.6. In the equivalent case, the equivalence can be proved in a short execution time. This is because only one extension of the verification areas was required to prove the equivalence, and the extended areas were still small. On the other hand, in the inequivalent case, we could not get the result within 12 hours. In this case, the number of the extensions is not so large. However, after extended four times, the verification areas become the whole designs. The main reason of this result is that the design has many conditional branches(actually, totally 364 branching points exist), and they are included after the extensions that extend backwardly along the control dependence. As a result, in the inequivalent case, the whole designs had to be symbolically simulated, which should take too long time.

Table 7.6: Experimental results on the elevator controller designs

Design1	Design2	Result	Time	# of ext.
Elv_org	Elv_rev1	Equivalent	1.8 sec	1
Elv_org	Elv_rev2	–	> 12 hours	4

7.3 Summary

In this chapter, case studies with two large example designs are introduced. Through the case studies, the following two things are confirmed.

- If the designs are equivalent, only a small number of ExSDG nodes (statements) and verification area extensions are required to be symbolically simulated.
- Even when the difference includes a few ExSDG nodes, almost the all ExSDG nodes (statements) have to be symbolically simulated if the difference is not equivalent, which can result in a significant increase of the verification time.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this thesis, formal equivalence checking of C-based system-level design descriptions are proposed. This equivalence checking is strongly required when designs are started from system-level, since a lot of manual refinements are carried out in system-level. The proposed methods are based on a formal technique called symbolic simulation, where variables and operations are treated as symbols. To realize the equivalence checking of large designs, the proposed methods utilize the difference between the design descriptions under verification.

In this thesis, the contributions include the following items.

- A method to sequentialize parallel behaviors to a sequential behavior that are equivalent to the all schedulings of the original parallel behaviors. In symbolic simulation, all possible schedulings are required to be checked for the equivalence. Therefore, equivalence checking cannot finish since the number of possible schedulings grows exponentially to the size of the design. By applying the sequentialization method before applying symbolic simulation, designs with parallel behaviors are translated into one sequential design, which can significantly reduce the symbolic simulation time. The method checks the possibility of sequentialization of two parallel statements that are dependent to each other by checking properties on the execution orders. If the statements are always executed in the same order, they can be sequentialized. Through the experiments, we show that large designs can be sequentialized in minutes.

- An efficient equivalence checking method that can efficiently generate EqvClasses is proposed. The method can reduce the number of calling the procedure to check the equivalence of variables and expressions. This is realized by generating an EqvClass when two variables can be decided to be equivalent based on the difference between the two descriptions and the dependence from the difference. Experimental results show that the number of calling the procedure is significantly reduced.

- An efficient equivalence checking method that locally applies symbolic simulation to the difference and its related portions. One of the most serious

problems in the previous methods is that symbolic simulation cannot be applied to whole of the design descriptions. This method reduces the problem by applying symbolic simulation locally to the difference, which results in the significant reduction of the paths to be symbolically executed. By the experiments, some practical optimizations can be verified with small local verification areas in short time.

- An equivalence checking method for loop optimizations without unrolling is proposed. When checking the equivalence of two loops by symbolic simulation, they have to be unrolled to avoid the paths of infinite length. Thus, the verification takes long time when the number of unrolling required is large. In the proposed method, analyzing data flow graph, the symbolic values of the loop iterators that are required to compute the output arrays. After that, symbolic simulation is applied only to the extracted values of the iterators and the equivalence is checked. Using this method, the experimental results have confirmed that the equivalence of loops between optimizations can be verified efficiently without entirely unrolling the loops.

- The proposed methods are implemented as a tool system called FLEC. FLEC is a framework for verification of system-level or RTL designs. In the tool, Extended System Dependence Graph (ExSDG) is used as an internal representation of designs under verification. The proposed method where symbolic simulation is applied locally to difference of designs has been implemented using ExSDG in FLEC. Through the case studies with two large designs, MPEG4 and an elevator controller, we can see that the equivalence can be proved in a short time. However, if the designs are not equivalent, the whole designs come to be symbolically simulated, which results in too long verification time.

8.2 Future Work

In this thesis, several methods for equivalence checking of system-level designs are proposed and the effectiveness of those methods has been confirmed through the experiments. However, there are several problems when those methods are applied

in practical design. As future work, the following issues should be studied to improve the performance of equivalence checking proposed in this thesis and support the practical use of the proposed methods.

- **Identification of more structural difference.** In this work, textual differences between two design descriptions are used to make the verification more efficient. However, textual differences are not meaningful when the difference is large or the coding styles are different. In such cases, more structural difference provides the essential difference between the designs, for example, the difference of two abstract syntax trees or system dependence graphs. We are planning to develop the procedure to identify the difference of two ExSDG in future.

- **Introduction of cutpoint.** In equivalence checking, the whole verification problem can be divided into small subproblems using cutpoints. A cutpoint in equivalence checking is a pair of two points from each design. By dividing the designs by cutpoints, equivalence checking becomes a set of small consecutive verifications.

- **Random simulation to identify potentially equivalent points.** To decide where a cutpoint should be, internal equivalent points play an important role, since we set a cutpoint to the variables that are equivalent with high probability. Such equivalent variables are called potentially equivalent points and can be detected by random simulation. Thus, a potentially equivalent points detection method assisted by random simulation is required to improve the efficiency more.

- **Propagation of equivalence for timed behaviors.** In this thesis, the methods are implemented as a tool FLEC. In FLEC, as a way to specify equivalence to be checked, users can specify the throughput and latency of the input and output variables. When timed behaviors are verified using local analysis as described in Chapter 5, the equivalence of local areas must be derived (i.e. what throughput and latency a local input/output variable should have?) from the top-level equivalence specification.

- **Automatic or semi-automatic deduction of the equivalence to be checked.** One of the problems in using the proposed verification is that users have to specify the equivalence to be checked before verification. It takes much effort especially when there are many input/output variables in designs or many different throughputs and latencies among those variables. To reduce the problem, automatic or semi-automatic deduction methods of the equivalence to be formally checked is strongly required in practical. There can be two ways to realize such methods. One is based on analysis of given designs, for example, by making structural correspondence between the designs. The other is based on traces generated by simulation. In the latter approach, potential equivalent points can be a good information also for the deduction of the equivalence to be checked.

- **Efficient implementation of symbolic simulation.** When symbolic simulation has to be applied to a whole design with a number of execution paths, the execution time increases exponentially with the design size. Therefore, more efficient implementations of symbolic simulation should be studied. There can be two ways to reduce the execution time of symbolic simulation. One is to divide the design under verification into small parts and symbolic simulation is applied to each of them. In this approach, symbolic simulation of each part can be run in parallel. The other is to run symbolic simulation of multiple paths at the same time. To realize this, the resulted symbolic equations of multiple paths must be kept together in a way where we can distinguish one execution path from another.

謝辞

この場を借りて、本博士論文を完成させるにあたり、お世話になった多くの方々に謝意を申し上げます。

本論文の執筆はもとより、卒論生の頃より 6 年もの長い期間に渡り、本研究やハードウェア設計の計算機支援全般、さらには、システムバイオロジーに至るまで、多岐に渡るテーマに関して多くの御指導・御助言をしてくださり、それらを通して、 国内外での学会発表、企業・大学の訪問やそこでの第一線で活躍する研究者を交えての研究交流、就業体験、大規模ツール開発体験などの他の研究室では決して経験できなかったであろう数多くの貴重な機会を提供してくださりました指導教官の藤田昌宏教授に深く感謝致します。

私が卒論生であった工学部 3 号館に研究室があった頃より同じ居室で研究室生活を見守ってくださり、検証の研究が御専門でないにも関わらず研究の相談にも快く応じていただき、ときにはおっしゃりづらい事柄であっても率直に助言をしてくださり、さらには、研究室の皆にとって有用な知識を提供してくださるなど、常に学生に近い立場で長い間に渡って私と研究室の学生の研究室生活を支えてくださりました小松准教授に深く感謝致します。

5 年半の長い間に渡って形式的検証の研究をともに行っていただき、特に並列処理の順序化の研究では具体的な定式化に至るまで御助言をいただきながら共同で進めて国際会議で発表できるような内容にまとめてくださり、また、研究以外の面でも頻繁に食事や飲み会に御一緒していただくなど、研究と私生活の双方において良き先輩であり続けてくださったサクンコンチャック タンヤパット氏に感謝致します。

研究室の先輩として常に真摯に研究に取り組む姿勢を見せてくださるとともに、研究室に在籍された時期には同じ博士課程の学生として研究の進め方などについて、さらには、御出身の中国四川省のいろいろな事柄について、気さくに話してくださった劉宇氏に感謝致します。

私が初めて国際会議へ発表へ行った際には一緒に徹夜で発表練習に付き合ってくださり、日本と異なって右側通行で走ってくる車に気付かずに道路を渡りそうになったときには間一髪で助けていただき、また、次の年のアメリカ出張では、寝ぼけてエレベータの非常ボタンを押したしまったところオペレータに英語で事情を説明してくださるなど、数々の危機から救っていただき、それ以上に、研究の進め方などについて親身に相談に乗ってくださりました瀬戸謙修氏に感謝致します。

研究分野は異なりますが、その高い見識に基づいて、研究を進めるにあたって有用ないくつかの助言を与えてくださり、また、新しく参加された研究室であるにも関わらず非常に多くの仕事をこ

なして研究室を支えてくださった吉田浩章氏に感謝致します。

研究を御一緒したのは少しの間だけでしたが、豊富な上位設計に関する知識や我々があまり聞くことのできない企業での設計の様子などを多く教えてくださった余宮尚志氏に感謝致します。

私が卒論生の頃より昨年まで長い間研究室の事務手続きを行っていただき、その間、私の出張手続きや出張中に生じた用件にも快く応じていただき、ときには研究室の引越しや掃除などにも参加して、研究室の雰囲気を和やかにしてくださった楯智子氏に感謝致します。

長い間に渡って研究室のいろいろな事務手続きを行っていただくとともに、研究室の図書リストの作成や故郷のお土産品などの差入れもしていただき、また、その堪能な英語力で国際会議の受け付けなどでも活躍してくださった寶田芽衣子氏に感謝致します。

事務補佐員として先生の教官室でお仕事をされているにも関わらず、研究室の学生と親しく接してくださった平出仙恵氏と船引美枝氏に感謝致します。

アメリカに滞在した際にはいろいろな名所に連れていっていただき、研究室においては、計算機環境の構築に力を尽くしてくださると同時に、FLEC の開発でも独自のポリシーを基に重要な役割を果してくださり、また、おいしいカキ飯の素を紹介してくださった小島慶久氏に感謝致します。

博士課程に入って、ほとんど何も知らない状態から研究を始められ、慣れない日本での生活の中でも懸命に、そして、前向きに研究に取り組んでいる姿勢を示してくださった高尚華氏に感謝致します。

研究についてはあまり話をする機会がありませんでしたが、折々に、故郷のインドの興味深い話をしてくれたり、インドのお菓子を分けてくれたりしたクリシュナムリティ ラトナ氏に感謝致します。

修士課程学生であるにも関わらず研究室の物品購入の手続きを行ったり、他の学生の計算機のセットアップをしてくれたりする一方で、それでもしっかり研究を進めて研究成果 STEP の完成に邁進してくれた石川悠司氏に感謝致します。

修士課程学生であるにも関わらず非常に高い金融に関する知識を有し、金融工学に関する研究論文を紹介してもらい、他方では、外部向けセミナーで講師を務めるなど研究面でも多いに活躍した安藤大介氏に感謝致します。

修士課程学生であるにも関わらず 2 年続けて異なる共同研究の中心となって研究を進めてもらい、自分の研究と直接には関係のないことでもセミナーへの参加などを通して意欲的に取り組むなど積極的に研究に取り組む姿勢を見せてくれた李蓮福氏に感謝致します。

まだ研究成果は多くはありませんが、留学生として、また、研究室の一員として共に研究に励んでいる高飛氏と許金美氏に感謝致します。

卒論生として研究を意欲的に進める一方で、研究室において「鍵をかう」という言葉を普及することに共に力を尽くした森下賢志氏に感謝致します。

研究室の OB として、今でも機会があればイベントに参加してくれる、卒論生時代の唯一の同期である吉田充孝氏と修士課程時代の唯一の同期である田辺健氏に感謝致します。また、同様に、卒業後も研究室を非常に頻繁に訪問して、私よりも研究室のイベント参加率が高いと思われる松井健氏

と佐々木俊介氏に感謝致します。

アメリカ滞在中には、毎週末、ただ渇いた喉を潤すためだけにアラメダとサンノゼの間を送迎してもらい、研究室においては、在学中に変換器合成の研究の基礎を残してくれた渡邊翔太氏に感謝致します。

先輩として、あまり研究面の面倒を見てあげることはできませんでしたが、立派に研究をして卒業し、それぞれの場所で活躍していると思われる、かつての卒論生である後藤真毅氏、アディヤスレン アルタンビレグ氏、村田裕之氏に感謝致します。また、研究員や研究生として研究室で活動され、様々な日常の会話などと通して研究室を国際色豊かな場所にしてくださった、シャンカー スバシュ氏、アリザデ ビジャン氏、アディカリ カマル氏、徐玲氏、何凱隆氏に感謝致します。

私が卒論生であった 1 年間しか御一緒することはできませんでしたが、折をみて声をかけてくださり、また、在学時に見せていただいた、明るい性格や研究に対する姿勢、研究室への貢献などで常に私の目標であり続けてくださった久保賢生氏に感謝致します。

本研究の初期段階で貴重な御助言をいただくとともに、形式的検証に関する幅広い内容について教えていただき、また、発表した論文の大半で共著者として研究の進展に貢献してくださり、研究以外の面でも様々な相談に親身に応じてくださった齋藤寛氏に感謝致します。

最後に、卒論生として配属されてから 4 年間、しっかりと形式的検証について自分で勉強して知識を身につけ、あまりに頼りない先輩でしたが、私の言うことにも理解を示してくれ、FLEC の実装に多大な貢献をし、そんな中でも自分の研究の方向性をしっかりと保ち続け、今後の研究室の検証の研究の中心となるであろう西原佑氏に感謝致します。

平成 19 年　12 月　13 日

Bibliography

[1] S. Abdi and D. Gajski, "A Formalism for Functionally Preserving System Level Transformations," *Proc. of Asia South Pacific Design Automation Conference 2005*, pp.139–144, January 2005.

[2] S. Abdi and D. Gajski, "Functional Validation of System Level Static Scheduling," *Proc. of Design, Automation and Test in Europe*, pp.542–547, March 2005.

[3] D. Ando, T. Nishihara, S. Sasaki, T. Matsumoto, and M. Fujita, "Design Error Detection in System-Level Designs by Dependence Analysis and Formal Checker," *Proc. of International Workshop on Logic and Synthesis 2006*, pp.255–262, June 2006.

[4] P. Ashar, A. Raghunathan, A. Gupta, and S. Bhattacharya, "Verification of Scheduling in the Presence of Loops Using Uninterpreted Symbolic Simulation," *Proc. of 1999 International Conference on Computer Design*, pp.458–466, 1999.

[5] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, Jan. 2000.

[6] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement," *Proc. of 12th International Conference on Computer-Aided Verification*, pp.154–169, July 2000.

[7] E. Clarke and D. Kroening, "Hardware Verificaiton using ANSI-C Programs as a Reference," *Proc. of Asia South Pacific Design Automation Conference*, pp.308–311, Jan. 2003.

[8] E. Clarke, D. Kroening, and K. Yorav, "Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking," *Proc. of 40th Design Automation Conference*, pp.368–371, 2003.

[9] D. W. Currie, A. J. Hu, S. Rajan, M. Fujita, "Automatic Formal Verification of DSP Software", *Proc. Design Automation Conference*, pp.130–135, June 2000.

[10] B. Dutertre and L. de Moura, "A Fast Linear-Arithmetic Solver for DPLL(T)," *Proc. of 18th International Conference on Computer Aided Verification*, pp.81–94, August 2006.

[11] D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publisher, March 2000.

[12] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL(T): Fast Decision Procedures," *Proc. of 16th International Conference on Computer Aided Verification*, pp.175–188, July 2004.

[13] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," *Proc. of 9th International Conference on Computer Aided Verification*, pp.72–83, June 1997.

[14] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations," *Proc. of 16th International Conference on VLSI Design*, pp.461–466, January 2003.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy Abstraction," *Proc. of ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, pp.58–70, 2002.

[16] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, Vol. 23, pp.279–294, May 1997.

[17] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, Vol.12, No.1, pp.26–60, 1990.

[18] C. Karfa, C. MAndal, D. Sarkar, S.R. Pentakota, and C. Reade, "A Formal Verification Method of Scheduling in High-Level Synthesis," *Proc. of 7th International Symposium on Quality Electronic Design*, pp.71–76, March 2006.

[19] T. Matsumoto, H. Saito, and M. Fujita, "An Equivalence Checking Method for C Descriptions Based on Symbolic Simulation with Textual Differences," *IEICE Trans. on Fundamentals*, Vol.E88-A, No.12, pp.3315-3323, Dec. 2005.

[20] T. Matsumoto, H. Saito, and M. Fujita, "Equivalence Checking of C Programs by Locally Performing Symbolic Simulation on Dependence Graphs," *Proc. of 7th International Symposium on Quality Electronic Design*, pp.370–375, March 2006.

[21] K. McMillan. *Symbolic Model Checking*, Kluwer Academic Publishers, 1997.

[22] M.H. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *Proc. of 38th Design Automation Conference*, pp.530–535, June 2001.

[23] 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," *Proc. of International Workshop on Logic and Synthesis 2007*, pp.83–90, July 2007.

[24] G. Ritter, "Formal Sequential Equivalence Checking of Digital Systems by Symbolic Simulation," PhD thesis, Darmastadt University of Technology and Universite Joseph Fourier, 2000.

[25] H. Saito, T. Ogawa, T. Sakunkonchak, M. Fujita, and T. Nanya, "An Equivalence Checking Methodology for Hardware Oriented C-based Specification," *Proc. of International Workshop on High Level Design Varidation and Test*, pp.139–144, 2002.

[26] T. Sakunkonchak, S. Komatsu, and M. Fujita, "Synchronization Verification in System-Level Design with ILP Solver," *Proc. International Conference on Formal Methods and Models for Codesign*, pp.121–130, July 2005.

[27] T. Sakunkonchak, T. Matsumoto, H. Saito, S. Komatsu, and M. Fujita, "Equivalence Checking in C-based System-Level Design by Sequentializing Concurrent Behaviors," *Proc. of IASTED 3rd International Conference on Advances in Computer Science and Technology*, pp.36–42, April 2007.

[28] L. Semeria and G. DeMicheli, "SpC: Synthesis of Pointers in C," *Proc. of IEEE/ACM International Conference on Computer-Aided Design 1998*, pp.340–346, November 1998.

[29] L. Semeria and G. DeMicheli, "Resolution, Optimization, and Encoding of Pointer Variables for the Behavioral Synthesis from C," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol.20, No.2, pp.213–233, February 2001.

[30] L. Semeria, A. Seawright, R. Mehra, D. Ng, A. Ekanayake, and B. Pangrle, "RTL C-Based Methodology for Designing and Verifying a Multi-Threaded Processor," *Proc. of 39th Design Automation Conference*, pp.123–128, 2002.

[31] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens, "Functional Equivalence Checking for Verification of Algebraic Transformations on Array-Intensive Source Code," *Proc. of Design, Automation and Test in Europe*, pp.1310–1315, March 2005.

[32] G. Stitt, F. Vahid, and W. Najjar, "A Code Refinement Methodology for Performance-Improved Synthesis from C," *Proc. of IEEE/ACM International Conference on Computer-Aided Design 2006*, pp.716–723, November 2006.

[33] A. Stump, C. Barret, and D. Dill, "CVC: a Cooperating Validity Checker," *Proc. of 14th International Conference on Computer-Aided Verification*, pp.500–504, 2002.

[34] 竹中崇, 向山輝, 若林一敏, 中田勝, 前川晃, 山際馨, "動作合成前後の動作記述とRTL 記述の論理等価性検証," 第 17 回回路とシステム軽井沢ワークショップ論文集, pp.555–560, 2004.

[35] M. Weiser, "Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction," PhD thesis, University of Michigan, 1979.

[36] SoC Environment: http://www.cecs.uci.edu/~cad/sce.html

[37] CodeSurfer: http://www.grammatech.com/products/codesurfer/

[38] SEMATECH: http://www.sematech.org/

[39] SystemC: http://www.systemc.org/

[40] CVC3: http://www.cs.nyu.edu/acsys/cvc3/

[41] CBMC Homepage: http://www.cs.cmu.edu/~modelcheck/cbmc/

[42] MPEG Software Simulation Group:
http://www.mpeg.org/MSSG/

[43] J. Daemen and V. Rijmen, *AES Proposal: Rijndael, Document Version 2*, September 1999.

Publications

Journal

1. Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita, "An Equivalence Checking Method for C Descriptions Based on Symbolic Simulation with Textual Differences," *IEICE Trans. on Fundamentals*, Vol.E88-A, No.12, pp.3315-3323, Dec. 2005.

International Conference

2. Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita, "Equivalence Checking of C-based Hradware Descriptions by Using Symbolic Simulation and Program Slicing," *Proc. of IEEE/ACM International Workshop on Logic and Synthesis 2003*, pp.252–259, May 2003.

3. Takeshi Matsumoto, Thanyapat Sakunkonchak, Hiroshi Saito, and Masahiro Fujita, "Verification of Behavioral Consistency in C by Using Symbolic Simulation and Program Slicer," *Proc. of International Conference on Dependable Systems and Networks 2003*, pp.w80–w84, June 2003.

4. Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita, "An Efficient Equivalence Checking of Similar C Descriptions with Use of the Textual Difference," *Proc. of IEEE/ACM International Workshop on Logic and Synthesis 2004*, pp.314–320, June 2004.

5. Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita, "An Equivalence Checking Method for C Descriptions based on Symbolic Simulation with Tex-

tual Differences," *Proc. of IASTED International Conference on Advences in Computer Science and Technology*, pp.246–251, Nov. 2004.

6. Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita, "Equivalence Checking for Transformations and Optimizations in C Programs on Dependence Graphs," *Proc. of International Workshop on Logic and Synthesis 2005*, pp.357-366, June 2005.

7. Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita, "Equivalence Checking of C Programs by Locally Performing Symbolic Simulation on Dependence Graphs," *Proc. of 7th International Symposium on Quality Electronic Design*, pp370–375, Mar. 2006.

8. Takeshi Matsumoto, Daisuke Ando, Tasuku Nishihara, and Masahiro Fujita, "Development and Verification of a Collaborative Printing Environment," *Proc. of IEEE 5th International Conference on Creating, Connecting and Collaborating through Computing*, pp.95–102, Jan. 2007.

9. Thanyapat Sakunkonchak, Takeshi Matsumoto, Hiroshi Saito, Satoshi Komatsu, and Masahiro Fujita, "Equivalence Checking in C-based System-Level Design by Sequentializing Concurrent Behaviors," *Proc. of IASTED 3rd International Conference on Advances in Computer Science and Technology*, pp.36–42, April 2007.

10. Takeshi Matsumoto, Kenshu Seto, and Masahiro Fujita, "Formal Equivalence Checking for Loop Optimization in C Programs without Unrolling," *Proc. of IASTED 3rd International Conference on Advances in Computer Science and Technology*, pp.43–48, April 2007.

Domestic Conference

11. 松本 剛史, 齋藤 寛, 藤田 昌宏, "C 言語でのハードウェア記述に対する効率的な等価性検証手法の提案," 電子情報通信学会技術研究報告 Vol.103, No.40, pp.31–36, 2003 年 5 月.

12. 松本 剛史, 齋藤 寛, 藤田 昌宏, “C 言語を対象とした記述間の差異に基づく効率的な等価性検証手法,” 電子情報通信学会技術研究報告, Vol.103, No.702, pp.61–66, 2004 年 3 月.

13. 松本 剛史, 齋藤 寛, 藤田 昌宏, “C 言語動作記述の既存 RTL 用検証ツールを用いた検証の提案,” DA シンポジウム 2004 論文集, pp.241–246, 2004 年 7 月.

14. 松本 剛史, 齋藤 寛, 藤田 昌宏, “C ベース高位設計における等価性検証フレームワークと反例解析手法の提案,” 第 18 回回路とシステム軽井沢ワークショップ論文集, pp.557-562, 2005 年 4 月.

15. 松本 剛史, 齋藤 寛, 藤田 昌宏, “依存グラフを用いた局所的な記号シミュレーションによる C 言語記述に対する等価性検証手法の提案,” 電子情報通信学会技術研究報告, Vol.508, No.58, pp.25-30, 2005 年 5 月.

16. 松本 剛史, 小松 聡, 藤田 昌宏, “動作合成前後の設計記述に対する記号シミュレーションによる形式的等価性検証の検討,” 電子情報通信学会技術研究報告, Vol.106, No.32, pp.7–12, 2006 年 5 月.

17. 松本 剛史 , Thanyapat Sakunkonchak, 齋藤 寛, 小松 聡, 藤田 昌宏, “線形計画法に基づく逐次化を利用したシステムレベル設計での動作並列化前後での等価性検証手法,” DA シンポジウム 2006 論文集, pp.157–162, 2006 年 7 月.

18. 松本 剛史, 瀬戸 謙修, 藤田 昌宏, “C 言語プログラムにおけるループ最適化に対するループ展開を伴わない等価性検証手法,” 電子情報通信学会技術研究報告, Vol.106, No.602, pp.55–60, 2007 年 3 月.