Formal Verification and Security Analysis of High-level Synthesis

Ramanuj Chouksey

Formal Verification and Security Analysis of High-level Synthesis

Thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

by

Ramanuj Chouksey

Under the supervision of

Dr. Chandan Karfa

Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Guwahati 781039, India
August, 2020

Declaration

I, Ramanuj Chouksey, confirm that:

a. The work contained in this thesis is original and has been done by myself and the general supervision of my supervisors.

b. The work has not been submitted to any other Institute for any degree or diploma.

c. Whenever I have used materials (data, theoretical analysis, results) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references.

d. Whenever I have quoted written materials from other sources, I have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

Place: IIT Guwahati
Date: 10 June 2020

Ramanuj Chouksey

Certificate

This is to certify that the thesis entitled "**Formal Verification and Security Analysis of High-level Synthesis**,", submitted by **Ramanuj Chouksey** to Indian Institute of Technology Guwahati, is a record of bona fide research work under my supervision and I consider it worthy of consideration for the award of the degree of Doctor of Philosophy of the Institute.

Dr. Chandan Karfa
Assistant Professor
CSE, IIT Guwahati

Date: 10 June 2020

Acknowledgements

First of all, I express my deep sense of regards and indebtedness to my supervisor Dr. Chandan Karfa for his valuable guidance, continuous encouragement and wholehearted support, which are of immense help to me in completing this thesis. Dr. Chandan Karfa is an excellent advisor. He taught me how to choose the problems to work on, how to write papers, and how to present them. Without his support and encouragement, this thesis would not have been accomplished. I would like to give the best "thank you" to Prof. Purandar Bhaduri for his valuable insights and comments on the translation validation part of this thesis. His deep theoretical knowledge had a great influence on my Ph.D research and future career. I would like to thank the rest of my thesis committee members: Prof. J. K. Deka, Prof. H. K. Kapoor, and Prof. K. V. Krishna for their insightful comments and encouragement. Their comments and suggestions helped me to widen my research from various perspectives.

I would like to express my heartful gratitude to the director, the deans and other managements of IIT Guwahati whose collective efforts has made this institute a place for world-class studies and educations. I am thankful to all faculty and staff of Department of Computer Science and Engineering for extending their co-operation in terms of technical and official support for the successful completion of my research work.

I am grateful to my parents for their unending love and support. It would be impossible to attempt to enumerate all those friends who have been supportive throughout my tenure as a PhD student. To all of you, I offer my sincere love, gratitude and appreciation. I would like to thank my wife Sivashankari for her unconditional love and making me feel at all times that my education and my dreams are as important as her own. I would like to thank my daughter Kumudini whose contagious smile can brighten up any tiring day. This thesis is dedicated to my loving wife Sivashankari.

Abstract

High-level synthesis (HLS) is the process of translating a behavioral description written in C/C++ into a Register Transfer Level (RTL) desgin. HLS tools are large and complex programs that may be incorrect in some contexts, which might introduce bugs in the generated RTL. Translation validation is the process of proving that the target code is a correct translation of the source program being compiled. In this thesis, a translation validation method based on propagation of mismatch values in a path-based equivalence checking method (PBEC) framework is proposed to validate the various scheduling optimizations during HLS efficiently. Specifically, this method verifies code motion involving loops, ignores the false computations, and handles the scenarios involving path merge/split. We have analyzed the correctness and complexity of the method. Experiments on various HLS benchmarks demonstrate the efficiency and scalability of our method.

In the case of non-equivalence, PBEC approaches provide too little information to debug the root cause of the non-equivalence. This thesis presents a counter-example generation framework to demonstrate the non-equivalence between the input behavior to HLS and the scheduled behavior generated by HLS. Equivalence checking of programs is an undecidable problem in general. Therefore, a PBEC method may produce a false negative results for which the counter-example will not arise. However, this helps the verification engineer to identify the limitation of the current translation validation tool and hence its enhancement in future.

Logic locking is an Intellectual Property (IP) protection technique against IP piracy, reverse engineering, hardware Trojans and counterfeiting attacks. RTL locking during HLS seeks to prevent IP theft of a design by locking the RTL description that functions correctly on the application of a key. This thesis introduces a satisfiability modulo theories (SMT) attack to determine the secret key of a locked RTL design. We have shown that our tool can detect keys of a locked RTL generated by TAO, a state-of-the-art HLS locking solution.

Keywords: Translation Validation, Equivalence Checking, Code Motion Transformation, Finite State Machine with Datapath (FSMD), Logic locking, RTL Locking, SMT attack.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With rapid growing complexity in the modern Very Large Scale Integration (VLSI) system, designing high-quality hardware at register transfer level (RTL) under seeking better productivity in less time and with lower cost is challenging. To achieve a better quality of implementations and shorter specification-to-product times of these microelectronic systems there is precisely need to perform design modeling, synthesis, and validation at higher levels of abstraction. The high-level behavioral specifications are simpler to write and to comprehend (and, therefore, update) and less error-prone. This significantly encourages the designer to design a complex system at a higher level of abstraction and uses High-level synthesis (HLS) [1–5] to generate RTL automatically from high-level behavioral description written in C/C++ or SystemC. The Behavioral description defines the design functionally at the high level of abstraction and thus allows concise, reusable, and readable design descriptions. The objective of HLS is to address the exacting demands to develop feature-rich, optimized, and complex hardware systems within aggressive time-to-market schedules. As shown in Fig. 1.1, HLS takes the high-level description of an application, executes several sub-tasks, and generates the RTL architecture. Typically the sub-tasks are:

1. **Preprocessing:** This task transforms the input description into an intermediate form more suitable for HLS, usually a control data flow graph (CDFG) [6]. In this step, several code optimizations are applied to improve the quality of synthesis results for designs. The common transformations applied during preprocessing are common sub-expression elimination (CSE), copy propagation, constant propagation, dead code elimination, loop invariant code motion (LICM) as well as restructuring transformation by function inlining and loop transformations (loop unrolling, loop fusion). These transformations increase the scope of parallelizing optimization in the scheduling phase that follows.

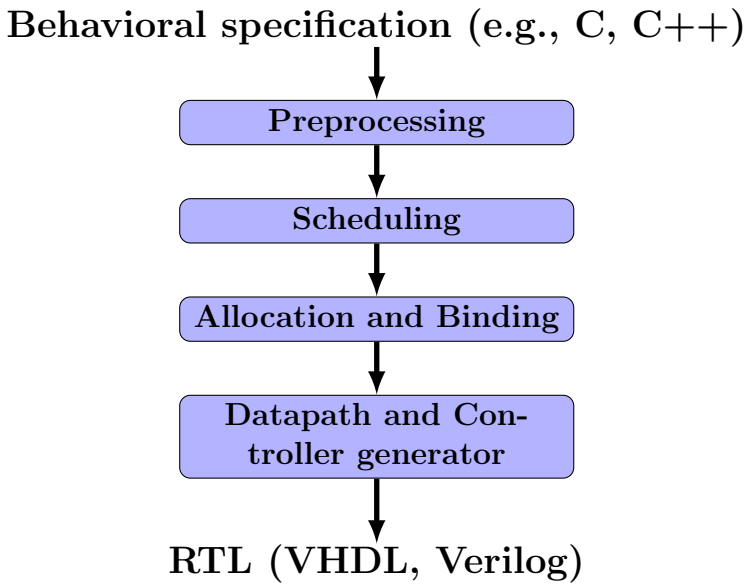


Figure 1.1: High-level synthesis flow

2. **Scheduling:** For untimed C/C++ designs, this step adds time to design and determines the time step or the clock cycle in which each operation of the design executes. The scheduling phase also employs a set of speculative, beyond-basic-block code motions to enhance concurrency and hence improving resource utilization [7–11]. The scheduling phase also applies additional transformations "dynamically" during scheduling such as dynamic common sub-expression elimination [12]. These dynamic transformations take advantage of changes created by speculative code motions.

3. **Allocation and Binding:** Allocation determines the type and quantity of resource storage and functional units, for each data object and operation of the input program. Binding assigns operations onto specific functional units. Binding assigns operations, variables that carries values across cycles, data structures, and data transfers onto specific functional units, storage elements (registers or memory blocks), and interconnections, respectively. In addition, several variables with mutually exclusive lifetimes are assigned to the same storage units.

4. **Datapath and Controller Generation:** This step generates a control unit that implements the schedule. This control unit generates control sig-

nals that control the flow of data through the datapath (i.e., through the multiplexers).

The latest HLS tools [13–18] are complex and use a variety of transformations to optimize the synthesis result for metrics like area, performance, and power. Ensuring the correctness of such transformations has become absolutely critical for the reliability of HLS tools.

1.1 Correctness of High-level Synthesis

HLS tools are usually very large and complex piece of software. They are prone to logical and implementation errors. In spite of rigorous testing, bugs in HLS tools may go unnoticed. A bug in an HLS tool can in turn introduce bugs in generated RTL. RTL designs with bugs have expensive outcomes if they go unnoticed until after production. Hence, the correctness of HLS has always been an important concern. Formal verification can be used to provide guarantees of HLS correctness. There could be two approaches for formal verification of HLS:

1. HLS tool verification
2. Translation validation

HLS Tool Verification

HLS tool verification includes techniques whose goal is to prove that HLS itself is correct. The primary advantage of this approach is that it can prove the correctness of the HLS tool once and for all, before they are run even once. However, these techniques that provide once-and-for-all guarantees require user interaction and immense manual effort. It also requires knowledge about internal algorithms used during HLS, which is often not available because most of the HLS tools are closed source [16–18].

Translation Validation

In general, it is hard to prove that an HLS tool with several hundred thousand lines of code always produces the transformed behaviors that are semantically

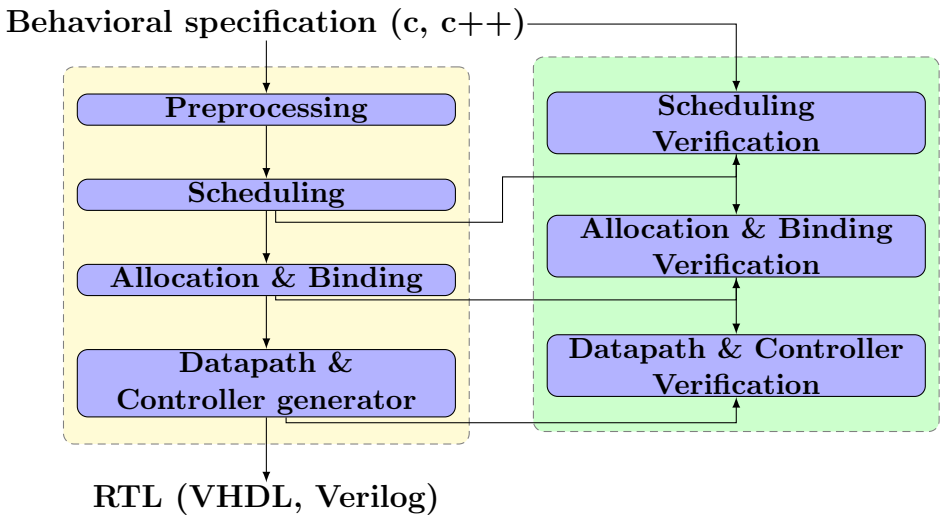


Figure 1.2: Phase-wise verification of HLS

equivalent to their source behaviors. The Translation validation [19–21] provides an alternative to prove semantics preservation for the transformations involved in HLS tools. In the translation validation approach, the HLS tool is not verified. Instead, it verifies the correctness of each run of the HLS tool. Translation validation techniques try to show for each translation that the HLS tool performs, that the transformed behavior generated by the tool is semantically equivalent to the source behavior. Even if this approach does not guarantee that the HLS tool is bug-free, it guarantees that any error in translation will be caught when the tool runs, preventing such errors from propagating any further in the hardware fabrication process.

Translation validation has previously been applied with success in the context of optimizing compilers [19, 21, 22]. Translation validation technique has been maturing via its use in verifying the correctness of the HLS. The existing works in translation validation of HLS can be divided into two categories:

1. End-to-end verification of HLS

2. Phase-wise verification of HLS

Because of the huge semantic gap between the source behavior and the generated RTL design, end-to-end translation validation techniques [23–28] fall short of meeting all the challenges posed by phase explicit technique. These techniques

need to make some assumptions with respect to synthesis flow. Therefore, phase-wise translation validation, as shown in Fig. 1.2 techniques that can deal with the difficulties of synthesis sub-task independently, are preferable for HLS verification. Specifically, the scheduling verification phase ensures that the compiler optimizations and the scheduling of operations do not change the functionality of the input behavior. The allocation and binding verification phase verifies the correctness of register sharing among the behavioral variables. The datapath and controller verification phase ensures that the correct functional unit has been chosen, the correct functionality of the functional unit has been chosen, the communication network has been correctly generated to allow the necessary data flow for a specified operation and the control signals have been assigned for each operation of the behavior.

1.2 Security in High-level Synthesis

The increasing cost of Integrated Circuit (IC) manufacturing has forced many semiconductor companies to go fabless over the years [29]. Such fabless companies design ICs and use offshore third-party foundries for manufacturing. This creates the following security threats [30, 31]:

- An attacker in the foundry or a rogue user can reverse engineer the functionality to steal the Intellectual Property (IP).
- An untrustworthy foundry can overbuild ICs for illegal sale.
- A rogue element in a foundry can temper the ICs to insert malicious circuits in the form of Hardware Trojans (HTs).

These security threats (also known as supply chain attacks) pose a significant economic risk to most IC design companies. One approach to thwart the aforementioned supply chain attacks is logic locking [32–34]. Logic locking inserts additional logic into a circuit, primarily in the gate-level design, to lock the functionality of the circuit with a secret key. The target chip produces the correct output only when the key inputs are correct, and such key values are not shared with the manufacturer.

As the complexity of ICs continues to increase, designers are moving to a higher level of abstraction to meet the growing challenges. HLS tools have made significant progress in the past few years and have been successfully used to improve design productivity by allowing designers to design systems faster at a high-level of abstraction. Addressing logic locking during HLS is an interesting approach to design and integrate solutions at higher levels of abstraction. TAO [35] is an exhaustive solution for algorithmic obfuscation during HLS. TAO extends the traditional HLS flow to produce the obfuscated RTL description and makes reverse engineering and hence the IP theft difficult.

1.3 Motivations and Objectives

The scheduling phase in addition to preprocessing is one of the central tasks in HLS which involves complex heuristics to ensure that the design being synthesized can meet the timing and resource constraints. In order to improve the quality of synthesized design in terms of timing and/or performance, the scheduling phase performs a variety of transformations. Hence, the scheduling phase is of the most error-prone parts of an HLS tool. Moreover, various compiler optimizations such as constant propagation, copy propagation, common sub-expression elimination, code motion, loop transformations, etc. are applied during preprocessing and scheduling to improve the synthesis results. A method for the fully automatic equivalence checking of a design before and after the scheduling step of HLS must be considered to ensure that the scheduling phase is correct.

The verification of the scheduling phase in HLS is an active domain of research for the last ten years. A translation validation approach is proposed in [36] to validate the scheduling results of the SPARK HLS tool [15] against the initial high-level program. However, to validate the transformed code with a loop structure, the existing approaches have to iterate over the loop to find the fixed point while such a process does not always terminate. Alternatively, many *path-based equivalence checking* (PBEC) approaches [37–42] have been proposed for verification of scheduling of HLS. These translation validation approaches are useful since they can verify that the correct code resulted from various compiler optimization techniques applied in each iteration of the scheduling phase of HLS without unrolling the loops. A PBEC approach based on value propagation [42], for example,

can verify the code motion across the loops. The primary focus of the existing PBEC approaches is ensuring that the data dependencies are not violated due to scheduling of operations and transformation of behaviors due to the application of various compiler optimizations [15] applied during the scheduling phase of HLS. Code motion based optimizations are used in the scheduling phase of HLS tools to improve the quality of synthesis results. Frequently, in the source behavior, there exist some computations within a loop body which produce the same results each time the loop is executed. These computations can be moved outside the loop body in order to achieve better design performance during the scheduling phase of HLS. The existing PBEC approaches fail to handle the case of loop invariant code motion. A computation is called a false computation if it never executes [43]. The PBEC approaches cannot identify false computations hence fails to ignore the false computation and produce false negative results. To improve the conditional hardware reuse in HLS, the optimization techniques such as in [44] split a path into multiple paths in the scheduled behavior. In this case, the existing PBEC approaches fail to handle the scenario where a path in a behavior is equivalent to the union of the paths in another behavior.

The PBEC approaches have been successfully applied for verification of the scheduling phase of HLS. These approaches can be sound but not complete [45]. Therefore, existing PBEC approaches may produce false negative results. The non-equivalence cases require further investigation of the two behaviors being compared by some human experts. In the case of non-equivalence, these approaches do not provide sufficient information for debugging the issue. A counter-example which will demonstrate the non-equivalence between the source behavior and the scheduled behavior generated by HLS will add significant value to the adoption of such PBECs. There is no work which generate a counter-example in the context of the path-based equivalence checking.

There is increasing security concerns such as hardware Trojans, IC counterfeiting, IP piracy, and unauthorized overproduction in the fabless mode of manufacturing. One approach to prevent from such attack is logic locking. Logic locking techniques hide the IC's functionality by manipulating the hardware description language. Starting with the SAT attack [46], the past few years have witnessed a flurry of activity on logic locking [34, 47–52], both on the attack and protection side. However, since the attack operates at the gate-level, these techniques are

not scalable to practical designs with hundreds of thousands of gates and flip-flops. Recent work has advocated for defenses that perform logic locking during HLS [35, 53]; the resulting RTL locked netlists are large and consequently less vulnerable to conventional gate-level SAT attacks.

With the above discussion, the following objectives were identified:

1. Translation validation of code motion transformations involving loops
2. Translation validation of code motion causing split/merged paths
3. Counter-example generation for PBEC approaches in case of non-equivalence
4. Evaluate the security of a state-of-the-art RTL locking scheme

1.4 Contributions of the Thesis

In the following, we outline in brief the contributions of this thesis on each of the objectives identified above.

1.4.1 Translation Validation of Code Motion Transformations Involving Loops during Scheduling

The primary contribution of this work is a PBEC approach based on value propagation, a translation validation approach, to verify the correctness of various optimization techniques applied during the scheduling phase of HLS. The behaviors are modeled as a finite state machine with datapath (FSMD) in our approach. Our approach breaks down an FSMD into smaller segments by introducing cutpoints so that each loop in the FSMD is cut by at least one cutpoint. This is based on the Floyd-Hoare method of program verification [54]. The set of all paths from a cutpoint to another cutpoint without any intermediate occurrence of a cutpoint is a path cover of the FSMD. Our approach establishes the equivalence between two behaviors by showing the equivalence between the paths present in the path cover of these two behaviors. A PBEC approach based on value propagation was proposed in [42] to verify code motion across loops but fails to show the equivalence when a code is moved outside the loop body from inside it in the transformed behavior. The proposed method can show the equivalence even some loop invariant

operation is moved before (after) the loop from inside it. This method is also capable of verifying the uniform code motion, non-uniform code motion techniques. In the presence of *false computations*, existing PBEC approaches produce a false negative result while this method is capable of avoiding these false computations. The presented method in this work establishes the functional equivalence between the result of scheduling and the behavioral specification of the design, using their FSMD models [1]. A notion of functional equivalence between two FSMDs has been defined, on the basis of which we verify the transformed behavior.

The correctness of the method has been proved and the complexity of the method has also been analyzed. Experimental results show the usefulness of the method. In particular, a bug in the HLS tool SPARK [15] involving loop invariant code motion is detected during the experiment.

1.4.2 Verification of Scheduling of Conditional Behaviors in High-level Synthesis

This work contributes a translation validation method to handle the scenario where a path in the source behavior splits into multiple paths in the scheduled behavior [44]. This work presents a notion of *split path equivalence* and introduces a new cutpoint selection scheme to show the equivalence for path split/merge scenarios. To identify the path split/merge scenario, the method tries to find a set of paths whose disjunction of the conditions of execution is equivalent to the condition of execution of a path in another behavior.

The method presented in Section 1.4.1 has been enhanced to handle the path split/merge scenarios. The enhanced method has been tested over the scheduled behavior obtained using Bambu HLS tools [14]. The scalability of the method has been checked over some larger CHStone benchmarks [55]. The formal proof of correctness of the method has also been presented. The computational complexity of the method is not worse than the methods proposed in [42, 56].

1.4.3 Improving Performance of a Path-Based Equivalence Checker using Counter-Examples

The contribution of this work is a counter-example generation mechanism for the PBEC approaches. This work shows how a counter-trace (*cTrace*) can be generated in non-equivalence cases reported by a PBEC approach. Using this *cTrace*, this work also presents a procedure to find suitable initialization values for input variables which reveal the non-equivalence (i.e., counter-example) by using Z3 satisfiability modulo theories (SMT) solver [57] or CBMC tool [58]. This counter-example generation mechanism improves the performance of a PBEC approach in the case of non-equivalence during verification of the scheduling phase of HLS. The counter-example generation mechanism also helps to identify some false negative cases of the PBEC approaches. The experimental results confirm that the PBEC approach is able to make stronger equivalence decisions with the help of a counter-example generation mechanism.

1.4.4 Security Analysis of Logic Locking during High-level synthesis

The contribution of this work is an SMT based algorithm to recover the secret keys of a locked RTL design. To the best of our knowledge, this is the first attack on RTL locking. The algorithm utilizes the rewriting method [59] to model an RTL design as a RTL finite state machine with datapath (RTL-FSMD). We abstract out the hardware information into a behavioral program on which we perform an SMT based attack. This SMT based attack is an oracle-guided attack. The incorrect keys are identified and eliminated using *distinguishing input patterns* (DIPs) [46].

A comprehensive evaluation of our attack algorithm has been conducted on locked RTL generated by TAO [35], a state-of-the-art RTL locking solution. Experimental evaluations show that our algorithm partially or completely break designs locked by TAO. In addition, the experimental evaluations also present that our approach is capable of attacking a locked C code. The strengths and weaknesses of our attack have been discussed and suggested some directions to design a secure RTL design.

1.5 Organization of the Thesis

The organization of the rest of this thesis is as follows:

Chapter 2 provides a detailed literature survey on state-of-the-art translation validation approaches to verify the correctness of HLS. It also presents a detailed survey on logic locking defenses and attacks.

Chapter 3 presents a translation validation approach based on value propagation for code motion involving loops. It also provides a solution to identify and ignore false computations during translation validation.

Chapter 4 identifies the limitation of existing path-based equivalence checking approaches to handle the control structure modification that occurs in the efficient scheduling of conditional behavior. It redefines the notion of the equivalence of paths in the context of path-based equivalence checking approach to handle the scenarios which involve path split/merge.

Chapter 5 presents how the equivalence information of a path-based equivalence checking method can be used to find a counter-trace in the case of non-equivalence reported by path-based equivalence checking methods. It also shows, for a given *cTrace*, how to find suitable initialization values for input variables which reveal the non-equivalence (i.e., counter-example) by using off-the-shelf SMT solvers [57] and CBMC tool [58].

Chapter 6 evaluates the security of a state-of-the-art HLS generated register RTL locking scheme using an SMT based algorithm to retrieve the secret keys. It demonstrates the attack on locked RTL generated by TAO [35], a state-of-the-art RTL locking solution. Empirical results show that it can partially or completely break designs locked by TAO.

Chapter 7 concludes and discusses some future research direction of this thesis.

Chapter 2

Literature Survey

In this chapter, we discuss some important research contributions on the verification of the HLS. This chapter also presents a comprehensive history of logic locking defenses and attacks. The objective of this study is to identify the prominent gaps in earlier literature which have been addressed in this thesis.

2.1 Verification of High-level Synthesis

In high level synthesis, a sequence of transformations is used to optimize the specifications at the behavior level into implementations at the register transfer level. HLS tools large and complex software developed over time by various developers on a legacy code-base. Therefore, the existence of bugs in some corner cases cannot be ruled out completely. Verifying the correctness of the generated RTL designs is, therefore, crucial to avoid substantial financial losses. The existing commercial HLS tools still use RTL co-simulation to validate the correctness of the generated RTL. However, simulation cannot guarantee the hundred percent correctness of the HLS tools. There are some efforts in research communities to develop formal verification of HLS. Since the semantic gap between RTL and the input C/C++ code is huge, end-to-end verification is difficult. Therefore, most of the existing methods try to verify a particular phase of HLS. In the following, we discuss the overall formal verification works of HLS. There may be two approaches for formal verification of HLS:(i) HLS tool verification, and (ii) Translation validation

2.1.1 HLS Tool Verification

This approach guarantees the correctness of the translation from high-level design to low-level design by proving the HLS tool itself correct. It proves the correctness of an HLS tool once and for all before it is ever run. Such effort is found in

CompCert compiler [60]. However, correctness by construction cannot be expected from a software system with several hundred thousand lines of code. Therefore, it is very hard to prove the correctness of the HLS once and for all. In fact, there is no such efforts reported in the literature for High-level Synthesis.

2.1.2 Translation Validation

Translation Validation [19] is a well-known way of increasing the reliability of HLS tools. In the translation validation approach, the HLS tool is not verified. Instead, a validator is associated with the HLS tool to verify the correctness of each run of the HLS tool. Translation validations aimed to check that each translation performed by the HLS tool preserves the semantics of the input behavior. The current works in translation validation of HLS can be separated into two classes.

1. End-to-end verification of HLS
2. Phase-wise verification of HLS

2.1.3 End-to-end Verification of HLS

An end-to-end verification approach finds equivalence between the behavioral description given as input (usually in C, C++, or SystemC) to any HLS tool and the RTL output of that HLS tool. The research reported in [24–28] tried to formally establish end-to-end equivalence between these two representations.

Radhakrishnan et al. [24] proposed a verification method using a witness generator. The method generates a sequence of elementary transformations that leads to the same effect as the applied HLS algorithm. If every transformation, identical in the derived sequence, is applied in the presence of a set of preconditions (which are proved to lead to a correct design), then the resulting RTL design is correct.

The authors in [26] proposed early cut-point insertion for checking the equivalence of high level software against RTL of combinational components. The basic idea is to derive an expression for both the C program and the RTL program, describing the input-output transition relation of the program and use symbolic execution and satisfiability solving to check equivalence between the two expressions. This paper only focused on combinational equivalence checking and did not address how to extend the proposed method for sequential equivalence checking.

Fujita [25] proposed a method based on virtual datapaths and controllers to verify equivalence between behavior level and RTL descriptions. First, a behavioral design is mapped to a virtual controller and virtual datapath, then equivalence of datapaths and the controller is established separately. However, in this work it has not been discussed how the equivalence checking works when the two descriptions are very different and cannot be mapped to the same datapath.

Leung et al. [27] proposed a translation validation technique for C to Verilog that establishes the equivalence between a C program and its Verilog counterpart without requiring any intermediate results from the HLS tool. They first convert both the C program and the Verilog program into a common intermediate representation (IR), then use bisimulation techniques to prove the two resulting IR programs equivalent. They invoke Daikon [61] to detect the likely invariants at cutpoints. However, in some cases, the likely invariants are not sufficient to prove post-conditions, and the algorithm will produce false negative results.

R. Mukherjee et al. [28] developed v2c, a tool that translates Verilog to C. The v2c accepts synthesizable Verilog as input and generates a word-level C program as an output. Equivalence checking is then achievable on C level with the help of either static analyzing tools or dynamic execution tools. They tried to apply v2c to generate equivalent C code from the RTL generated by a commercial HLS tool. However, they found that it cannot correctly map the co-ordination between the controller FSM and operations in blocks. Therefore, a formal equivalence proof is needed between the RTL and v2c generated C code. Thus, applying v2c based framework is not a natural solution for HLS functional verification.

Due to the optimizations performed at various stages of the synthesis process, the resulting RTL design bears little similarity to its specification. An end-to-end verification method for HLS is very tough and also inadequate for locating the exact sources of errors. Therefore, an end-to-end equivalence checker that can handle the complexities of modern day HLS tools is still not available.

2.1.4 Phase-wise Verification of HLS

The large difference in abstraction between the input behavior and RTL design makes end-to-end verification approach non-trivial. Therefore, the phase-wise verification technique which can handle the difficulties of each synthesis sub-task

separately is desirable for HLS verification. In the following, the verification of different subtasks of HLS is discussed.

Scheduling Verification

One of the most error-prone parts of an HLS tool is its scheduling phase since it performs aggressive optimizations to meet timing and resource constraints. Hence, it is necessary to validate the functional equivalence between the input behavior to HLS and the scheduled behavior generated by HLS.

Anderson [62] reported an early effort on the verification of as soon as possible (ASAP) scheduling transformation using theorem proving. The paper [63] identified a set of assertions and invariants that should be held at various steps of HLS. These invariants were inserted inside the implementation of the force-directed list scheduling (FDLS) algorithm to detect and isolate the errors in a specific run of the tool. The correctness of the FDLS algorithm is proved using the prototype verification system (PVS) theorem prover. In [64] scheduling results are verified based on precondition-based correctness and completeness of register transfer split.

Eveking et al. [65] represented the pre-scheduled and post-scheduled behaviors in the language of labeled segments (LLS) and developed the basic transformations to prove the computational equivalence of LLS. However, none of these techniques can verify code motion applied during the scheduling phase of HLS.

A formal verification of the scheduling phase of HLS using the FSMD model is reported in [66]. In this paper, cutpoints are introduced to construct the path cover for each FSMD. Each path of one path cover is then shown to be equivalent to some path of the other path cover. However, the technique presented in [66] assumes that during the synthesis process, the path structure of the input behavior is not modified and operations are not moved from one synthesis basic block to another. The authors extended their work in [37] to verify speculative code motions by concatenating critical paths. In this paper, the equivalence conditions are formulated in high-order logic, and used PVS theorem prover to verify their correctness. The method presented in this paper fails if the scheduler applies the non-uniform code motion transformations.

Karfa et al. proposed an equivalence checking method for verification of scheduling in [38]. In this work, an initial path cover is obtained by introduc-

ing cutpoints in the FSMD. The paper proposed a bisimulation based symbolic equivalence of the path covers of two FSMDs. During equivalence checking, a novel path extension method is proposed to dynamically remove some cutpoints to prove the equivalence. The work presented in this work takes care of both run time of the equivalence checker and the wider range of optimizations applied during scheduling. The method is applicable even when the scheduler changes the basic structure. This method works only for uniform code motion techniques. The paper [40] improved the equivalence checking method presented in [38] to deal with code transformations employing speculation and global common subexpression extraction.

The paper [39] improved the equivalence checking method presented in [38] to handle both uniform and nonuniform code motions applied during the scheduling phase of HLS. This work identified certain data-flow properties that must hold on the initial and the scheduled behaviors for valid nonuniform code motions. These properties are based on the definition-use chain [67] of the variables in the behavior. These properties are encoded as simple CTL (Computational Tree Logic) [68] formulae and invoke the model checking tool NuSMV [69] to verify them. The paper [41] uses machine learning (ML) techniques to recognize the corresponding path-pairs of FSMDs and reduces the complexity of the path-based FSMD equivalence checking problem.

The methods proposed in [37–41, 66] decompose each behavior into a finite set of finite paths. Equivalence of the behaviors is established by showing path level equivalence between two behaviors modeled as FSMDs. The transformation which modifies the control structure of the input behavior are handled through path extension. However, a path cannot be extended across a loop by definition of path cover. Therefore, all these methods fail to handle the transformations that result in code motion across loops, i.e., some code segment before a loop body is placed after the loop body, or vice-versa

The technique presented in [70] handles code motion across loops but it requires additional information from the synthesis tool that is difficult to obtain in general. The paper [42] introduced a notion of value propagation and widen the scope of the path-based mechanism [38] to handle code motion across loops. The paper [42] proposed a value propagation based equivalence checking (VP) method which also handles code motion across loops. This VP method is also capable of handling

control structure modification of input behavior and uniform and non-uniform code motion. Unlike the technique presented in [70], the VP method does not require additional information from the synthesis tool.

A translation validation approach is proposed in [36, 71] to validate the result of HLS against the initial high-level program. The method presented in these papers uses a bisimulation relation approach to prove the equivalence of two descriptions before and after the optimization carried out by the SPARK tool [15]. An improved translation validation of HLS proposed in [72] reduces the number of queries to an automatic theorem prover, such as Z3 [57], when compared with the method presented in [36, 71]. All these methods [36, 71, 72] are suitable for handling structure preserving transformations, while in HLS the structure may not be preserved in the case of path-based scheduling. The translation validation method proposed in [73] can deal with structure preserving and non-structure preserving optimization due to path-based scheduler [74]. However, to validate the transformed code with loop structure, the existing approaches have to iterate over the loop to find the fixed point while such a process does not always terminate. The paper [75] proposed an equivalence checking approach which combines translation validation with methods based on the cut-points, and shared value graphs (SVG) to handle various scheduling optimizations. The proposed method [75] uses the SVG technique to validate a predicate in one-pass and avoids the "may not terminate" problem of existing methods.

The paper [76] presented a method based on symbolic simulation together with identification and inductive verification of loop structures to verify compiler transformation commonly applied during the scheduling of HLS. It uses a symbolic execution technique to explore the paths of the input and transformed behaviors. It handles the path explosion and path explosion and non-termination in symbolic simulation issues through compositionality and cut-loop optimization. The paper [77] presented a scalable equivalence checking algorithm for validating scheduling transformations. The paper [77] validates various I/O timing modes such as cycle-fixed mode, superstate-fixed mode and free-floating mode. However, the algorithms presented in [76,77] can only compare two intermediate representations (IRS) that are structurally close. If a transformation significantly transforms the structure of an IR then the heuristics for detecting corresponding variables between the two IRs will not succeed, causing equivalence checking to fail.

Allocation and Binding Verification

The verification of allocation and binding phase verifies the functional unit allocation and binding and also verifies the register sharing among the behavioral variables.

Ashar et al. [78] proposed a complete procedure for verifying register-transfer logic against its scheduled behavior in a high-level synthesis environment. In Ashar et al., the verification task is partitioning into two subtasks, verifying the validity of register sharing and verifying correct synthesis of the RTL interconnection and control. The paper performs equivalence checking between behavioral specification and RTL implementation of designs by model checking.

The paper [79] reported a post-synthesis methodology based on theorem proving for formally verifying the various register allocation schemes. In this work, the scheduled and the RTL description are encoded as extended finite state machines (EFSMs). The method consists of determining the equivalence of critical states, critical variables, and critical paths of two EFSMs. However, in the presence of loops in the behavior, one may encounter an infinite number of execution paths from the initial state while showing the equivalence between two critical states.

The work proposed in [80] handles the high-level verification in two steps: verification of scheduling and verification of allocation and final architecture generation. The paper mainly proves that the final architecture consisting of a controller and a datapath is correctly generated from the abstract FSM obtained after the scheduling step. The approach, however, ignores register sharing verification. The prototype tool presented in this paper has no general inference rule to prove two algebraic expressions equal.

Karfa et al. [81] proposed a formal methodology for verifying the correctness of register sharing. The method models the behavior before and after the datapath synthesis as FSMDs and checks the equivalence of both FSMDs. The method is independent of the schemes used for register optimization. The method also works for both data intensive and control intensive input specification.

Datapath and controller verification

The datapath and controller verification ensures the correctness of the data-path interconnections and the controller. The paper [59] proposed a formal verification

method of the datapath and controller generation phase of a high-level synthesis process. This paper presents a rewriting method to obtain the register transfer operations executed in the datapath for a given control assertion pattern in each control step. It uses a state-based equivalence checking methodology to verify the correctness of the controller behavior. Some of the allocation and binding verification methods [78,80] treat the allocation, binding and the data-path and controller generation steps into one by verifying the final RTL against the scheduled behavior.

2.1.5 Our Objective

As discussed above, most of the existing works target the verification of the scheduling in HLS since the verification of this phase is the most challenging among all phases of HLS. In this thesis, we are also interested in verifying the pre-synthesis and the scheduling phases of High-level Synthesis. Code motion based optimizations are used in the scheduling phase of HLS tools to improve the quality of synthesis results in terms of timing performance. All of the above mentioned techniques fail to handle the scenario where some loop invariant operation is moved before (after) the loop from inside it. In the presence of false computations, these methods produce false negative results. These methods also fail to handle the scenario where a path in the source behavior splits into multiple paths in the scheduled behavior. In this thesis, we propose an equivalence checking method that verifies code motion involving loops, ignores the false computations, and handles the scenarios involving path merge/split along with uniform and non-uniform code motions and transformations which alter the control structure of a given behavior.

2.2 Logic locking: Defenses and Attacks

Logic locking is a technique that protects a hardware design netlist against the untrustworthy IC supply chain. Logic locking hides the functionality of a design by adding additional gates into the original design. Many logic locking techniques as well as attacks have been widely investigated for a decade. In this section, we discuss the existing works on logic locking both on attack and defense sides in

chronologically.

RLL [82] and FLL [83, 84]

Logic locking was first introduced in *EPIC*, which abbreviates "Ending Piracy of Integrated Circuits", [82]. EPIC used a random XOR/XNOR key gates insertion policy referred to as *random logic locking (RLL)*. It obfuscates the design by inserting XOR/XNOR key gates at a random location in a netlist; only a correct key makes the design to produce correct outputs. The drawback of RLL is that it does not necessarily ensure that the wrong keys corrupt the output. Consequently, an RLL netlist may produce correct output even for incorrect key values. Aims at overcoming the shortcoming of RLL, Rajendran et al. used fault simulation techniques in [83, 84] to guide XOR/XNOR key gates insertion. The proposed *fault analysis based logic locking (FLL)* uses a new insertion criterion called the *fault impact*. FLL inserts the key gates at the locations that exhibit the highest fault impact.

Sensitization Attack [85]

After introducing EPIC, Rajendran et al. [85] proposed a sensitization attack. Sensitization attack, which is an oracle-guided attack, tries to propagate a single key value to the output. The attacker analyzes the locked netlist and computes attack patterns that can sensitize individual key bit values to primary outputs. By applying these patterns to functional IC, the attacker observes and records this output as the value of the sensitized key-bit. The effectiveness of the sensitization attack depends on the location of the key gates. The sensitization attack is highly effective when key gates do not interfere with each other.

Strong Logic Locking [86]

Both RLL and FLL remain vulnerable to the sensitization attack. To prevent the sensitization attack, strong logic locking (SLL) was introduced. SLL inserts key-gates in a way that key-gates protect one another [86]. SLL inserts pairwise secure key gates that protect one another in a netlist. In SLL, it is not possible to sensitize the key-bit values to a primary output.

Hill-climbing Attack [87]

Plaza and Markov proposed the hill-climbing attack [87]. The hill climbing attack exploits test data to determine the secret key. The attack relies on a hill-climbing search algorithm which uses Hamming distance as a guiding metric. At first, the attack makes a random initial guess for key K_{rand}. The initial Hamming distance HD_{rand} is then computed between the test response and the locked circuit outputs corresponding to test stimuli. A randomly selected key bit in K_{rand} is then inverted and Hamming distance $\texttt{HD}_{\texttt{inv}}$ is computed. The HD_{rand} and $\texttt{HD}_{\texttt{inv}}$ are then compared. If $HD_{inv} < \texttt{HD}_{\texttt{rand}}$ then toggle is retained in K_{rand}. The inversion process is repeated until a key value K_{rand} is found that leads to a Hamming distance of zero. The attack is successful when $HD_{rand} = 0$. The hill climbing attack can successfully break RLL and FLL but it loses its effectiveness against SLL. The complexity of this attack quickly increases with an increasing number of key gates.

LUT based [88] and Weighted logic locking [89]

Apart from RLL, FLL and SLL, important research efforts on logic locking include a *look-up table (LUT) based locking* [88] and *weighted logic locking* [89] to find suitable key gate locations. The main objective of all these logic locking techniques is to increase the output corruptibility (i.e., produce more incorrect outputs for more input patterns) given an incorrect key. In 2015, Pramod et al. [46] developed a powerful attack that broke all logic locking techniques that existed at that time. The attack employs a Boolean satisfiability (SAT) formulation to encode of finding the logic locking key and commonly refer to it as the *SAT attack*. All the logic locking methods discussed above also remain vulnerable to Boolean Satisfiability based attack called SAT attack [46]. The SAT attack can easily break these logic locking techniques within a few hours even for a reasonably large number of keys.

SAT Attack [46]

The SAT attack is an oracle-guided attack. The SAT attack employs a SAT solver to find distinguishing input patterns that refine key search space iteratively. A DIP is an input value x_d for which at least two different key values, k_1 and k_2, produce differing outputs, o_1 and o_2, respectively. Since o_1 and o_2 are different,

				Output Y for different key values							
a	b	c	Y	k_0	k_1	k_2	k_3	k_4	k_5	k_6	k_7
0	0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0	0
0	1	1	1	1	1	1	0	1	1	1	1
1	0	0	0	0	0	0	0	1	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	0	1	1
1	1	1	1	1	1	1	1	1	1	1	0

Table 2.1: Maximal resilience against the SAT attack can be achieved by controlling the discriminating ability of input patterns

at least one of the key values is incorrect. A single DIP may rule out multiple incorrect key values. The SAT attack terminates when no more DIPs can be found, which means that the remaining key is guaranteed to be the correct key. The computational effort of the SAT attack depends on the order of choosing the DIP. However, the SAT attack selects the DIPs on a random basis. The larger the number of incorrect key values ruled out per DIP, the less the DIP required for the attack, which suggests a littler execution time of the attack.

Note that the worst case scenario for the SAT attack arises when it can eliminate at most one key for every DIP. In Table 2.1, a, b, c are the primary inputs, and Y is the corresponding primary output. It can be observed that the correct key value is $k6$. It also evident from Table 2.1 that in each row, there is at most one key value that generates an incorrect output. In such a scenario, the SAT attack with k key bits would require at least $2^{|k|} - 1$ DIPs. To prevent SAT attack, the first attempted approach was to reduce the number of wrong keys that each DIP can rule out. *SARLock* [34], *Anti-SAT* [50, 90], and *AND-tree insertion* [91, 92] can accomplish this.

SARLock [34]

SARLock, which abbreviates, "SAT Attack Resistant Logic Locking," thwarts SAT-based attacks by minimizing the number of keys that are ruled out by a single DIP. To accomplish this effect, SARLock integrates a comparator and a mask block with the original circuit. The comparator circuit which mimics Table 2.1 flips the

circuit output for only one input pattern for a given (wrong) key. The resulting locked circuit achieves the desired resistance against the SAT attack at minimal overhead. A small mask logic is inserted to prevent the assertion of the flip signal when a correct key is given. For each incorrect key value, an error is injected into the circuit for only one input pattern, leading to an incorrect output for the specific pattern. Assuming that $F(I)$ denotes the original circuit, the output O of the circuit locked using SARLock can be presented as $O = F(I) \oplus ((I == K) \oplus (I == k_s))$, where K denotes the key inputs, and k_s is the correct key value. For a key size k, the SARLock protection circuit consists of $k + 1$ 2-input XOR/XNOR gates and $2k + 1$ 2-input AND gates.

Anti-SAT Logic Locking [50, 90]

In Anti-SAT [50, 90], an Anti-SAT block comprises two blocks $B_1 = g(X, K_{11})$ and $B_2 = \overline{g(X, K_{12})}$. Both blocks share the same inputs X and are locked with different keys K_{11} andK_{12}. The one-bit output Y is the AND operation of the outputs of B_1 and B_2 blocks. The output Y is connected to the original circuit using an XOR gate. The functionality of the two blocks is complementary. When the correct key value is applied, for all inputs, $Y = 0$, leading to a correct output. If the incorrect key is applied, the output of B_1 and B_2 is 1 for a specific input pattern; for that pattern, $Y = 1$, and thus produce a fault in the original circuit. With Anti-SAT, only 1 key value among all wrong key values could be ruled out at each iteration of the SAT attack.

AND-tree Insertion (ATI) Logic Locking [91, 92]

While SAR-Lock and Anti-SAT add an external circuit to the original netlist AND-tree insertion (ATI) finds an AND-tree inside the original netlist and thus decrease the implementation overhead. The inputs of the identified AND-tree are camouflaged by inserting INV/BUF camouflaged gates. The INV/BUF gates can be replaced with the XOR/XNOR counterparts to obtain a logic locked AND-tree. An existing AND/OR tree can be identified by running a breadth-first search on the netlist. The SAT attack resilience of ATI grows exponentially with increasing key size, similar to that for SARLock and Anti-SAT. The major drawback of ATI is that it can only protect the parts of a circuit where the desired AND/OR trees

are present inherently. It does not offer a designer the flexibility to choose the logic to be protected.

Compound Logic Locking

The main drawback of the SARLock, AntiSAT, and ATI logic locking techniques is their low output corruptibility. Compound logic locking technique combines a low output corruptibility technique (e.g., SARLock, AntiSAT or ATI) with a high output corruptibility technique (e.g., RLL, FLL or SLL). For example, Compound techniques that improve the output corruptibility remain vulnerable to the approximate attacks. In [34] SARLock is combined with SLL, and in [90] AntiSAT is integrated with FLL.

Signal Probability Skew (SPS) Attack [48]

SPS breaks Anti-SAT. SPS exploits structural traces in the netlist to identify and remove the basic (unobfuscated) Anti-SAT and retrieve the original circuit within minutes. The attack uses the notion of signal probability to identify the output gate of Anti-SAT. The signal probability skew(SPS) is given as,

$$SPS(x) = Pr[x = 1] - 0.5 \tag{2.1}$$

where, $Pr[x = 1]$ denotes the probability of signal x being 1. For a signal y that is rarely 1, e.g. the output of a large AND tree, $SPS(y) = -0.5$. The absolute difference of the probability skew (ADS) of all gate outputs in the netlist are calculated and gate with the maximum ADS value is suspected to be the output Y of the Anti-SAT block. The signal Y is set to the value it is most skewed towards, thus defeating the protection offered by the SAT resilient block. SPS attack is scalable to large circuits and it becomes more effective with increasing key size. However, the attack becomes less effective in the presence of structural/functional obfuscation.

AppSAT Attack [93]

Shamsi et al. proposed an approximate attack based on the SAT attack and random testing (AppSAT) [93]. AppSAT aims at reducing a multi-layered defense

to single-layer (e.g., Anti-SAT+FLL to Anti- SAT). The SAT attack terminates when there is no DIP and reports the correct key but the AppSAT attack terminates when the Hamming distance between the correct output from the functional IC and the locked netlist is very low. Otherwise, random testing that resulted in a disagreement will be added to an SAT formula as a new constraint. Upon termination, the attack returns approximate correct key values, which results in an approximate netlist.

Double-DIP [94]

Shen et al. proposed in [94] the Double DIP SAT-based attack to reduce a compound logic locking technique to its low-corruptibility component. Similar to AppSAT, Double-DIP is an approximate attack. Double-DIP used 2-DIPs, which can eliminate at least two incorrect key values in a single iteration. Double-DIP attack terminates when 2-DIPs can no longer be found.

AppSAT-Guided Removal (AGR) Attack [48]

The AGR attack targets compound logic locking, particularly Anti-SAT + traditional logic locking. Unlike AppSAT, the AGR attack recovers the correct key. This attack integrates AppSAT with a simple structural analysis of the locked netlist. Firstly AppSAT is used to find the key of dedicated to the traditional locking technique. Then, a structural analysis of the Anti-SAT block allows discovering the last gate of the block.

Bypass Attack [47]

The Bypass attack uses a bypass circuitry around a locked netlist to nullify the error introduced by the locked circuit, thus restore its correct functionality. This attack is efficient against Anti-SAT and SARLock, even coupled with a traditional logic locking technique.

Tenacious and Traceless Logic Locking (TTLock) [51]

Both SARLock and Anti-SAT are vulnerable to removal attacks because they implement the original function. TTLock is an improvement of SARLock that

prevents a removal attack. TTLock modifies the original logic cone for exactly one input pattern. For this input pattern, the modified netlist and the original netlist differ in their outputs for all wrong keys. TTLock adds a comparator block to restore the correct functionality only for the correct key. Upon removal attack, the attacker still gets netlist which is different than the original one. However, in TTLock output differs from the original circuit for exactly one cube which results in low output corruptibility and is vulnerable to approximate attacks such as AppSAT.

Stripped-Functionality Logic Locking [52]

Stripped-functionality logic locking (SFLL), a logic locking technique that provides provable security against SAT, removal, and approximate attacks. SFLL has three variants: SFLL-HD, SFLL-flex, and SFLL-fault. SFLL-HD creates a functionality-stripped circuit (FSC) by inverting the output of the original circuit for $\binom{k}{h}$ input patterns that are of Hamming distance h from the k-bit secret key. With increasing h, the number of protected patterns increases binomially. For $h = 0$, SFLL-HD is equivalent to TTLock. In SFLL-flex, the designer choose the protected patterns freely irrespective of any key and hamming distance. SFLL-fault subtracts the logic by inserting fault injection. SFLL-fault, thus, does not leave any structural traces any traces for an attacker to exploit.

Functional analysis attack on logic locking (FALL) [49]

FALL attack uses structural and functional analyses of circuit nodes to first identify the gates that are the output of the cube stripping module to determine the locking key. The Functional analysis attack on logic locking (FALL) breaks SFLL-HD and SFLL-flex as well. However, the FALL attack cannot break SFLL-fault.

2.2.1 Our Objective

These attacks and defenses focus on the gate-level abstraction and have been demonstrated on small circuits like the ISCAS benchmarks. Recently, there has been an attempt to perform logic lock at the RTL [35, 95] and the C level [96]. TAO [35] is an example of such a scheme. However, to the best of our knowledge,

hardware security during HLS has not been studied. The SMT attack proposed in Chapter 6 is the first one on locking during HLS. While SMT has been used to unlock gate-level netlists [97], these methods do not apply to RTL unlocking.

Chapter 3

Verification of Code Motion Transformations Involving Loops during Scheduling

3.1 Introduction

3.1.1 Code Motion Techniques

In the scheduling phase, HLS tools enhance concurrency and hence improving resource utilization by moving operations across basic block boundaries, which is called code motion. In the next subsection we explore a set of speculative code motions that are useful for HLS. These code motions have been proposed for improving synthesis results in designs with complex control flow.

Speculative Code Motions

The speculative code motions enable movement of operations through, beyond, and into conditionals with the objective of extracting the inherent parallelism in design. Effectively, these code motions reorder operations to reduce the impact of choice of control flow in the input behavior. The speculation comes into four forms: (1) speculation, (2) reverse speculation, (3) conditional speculation, and (4) early condition execution. An overview of the various speculative code motions is shown in Fig. 3.1.

Speculation

Speculation refers to the unconditional execution of operations that were originally supposed to have executed conditionally. In this approach, the result of a

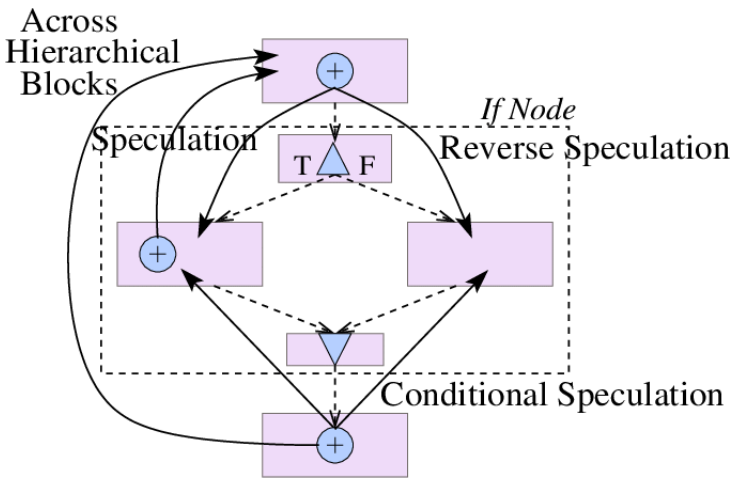


Figure 3.1: Various speculative code motions [15]

speculated operation is stored in a new variable.

Reverse Speculation

In reverse speculation operations before conditional block are moved into subsequent conditional block and executed conditionally. Reverse speculation has been variously referred to as lazy code motion or execution and duplicating down in past literature [7, 98].

Conditional Speculation

In conditional speculation an operation from after the conditional block may be duplicated up into preceding conditional branches and executed conditionally. This is similar to the duplication-up code motion used in compilers and the node duplication transformation discussed in [99].

Early Condition Execution

Early condition execution evaluates conditional checks as soon as possible. Reverse speculation can be coupled with early condition execution i.e., conditional check is moved up and all operations before the conditional block are reverse speculated into the conditional block.

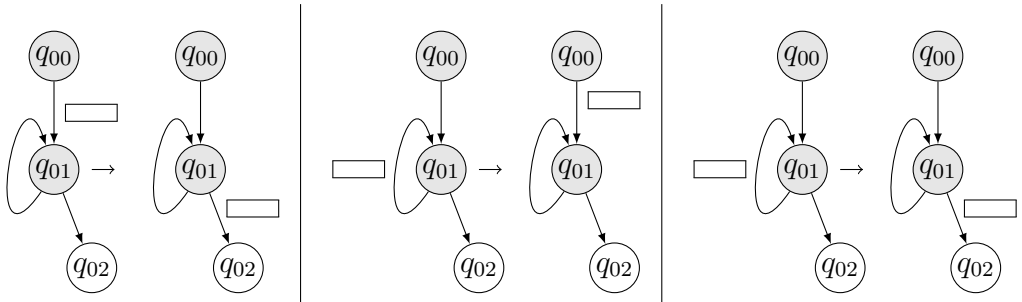


Figure 3.2: Three possible scenarios during code motion transformations involving loops

Loop Invariant Code Motion

Loop invariant code inside a loop body consists of statements or expressions which produce the same result each time the loop is executed. In other words, these statements are not dependent on loop iterations. This code can be moved outside the loop body without changing the program semantics. Loop invariant code motion improves overall program execution time by reducing the number of times loop invariant expressions are executed by a factor equal to the loop size.

As shown in Fig. 3.2, there are three possible scenarios during code motion transformations involving loops:

S_1 : Some code segment before a loop body is placed after the loop body or vice versa (i.e., code motion across loops).

S_2 : Some code segment is moved before the loop from inside the loop body.

S_3 : Some code segment is moved after the loop from inside the loop body.

Code motion based optimizations are used in scheduling phase of HLS tools to improve the quality of synthesis results. Code motion techniques change the data-flow of a behavior considerably. Therefore, it is necessary to verify the semantic equivalence between the original and the transformed behaviors.

3.1.2 Summary of Verification of Code Motion

Verification of code motion transformations has been an active research area for the last ten years [36–40, 42, 70, 75]. The methods [36–40, 75] fail to handle the case of

code motion across loops and loop invariant code motion. The technique presented in [70] handles code motion across loops but it requires additional information from the synthesis tool which is difficult to obtain in general. The VP method was proposed in [42] which also handles code motion across loops. Unlike the technique presented in [70], the VP method does not require additional information from the HLS tool.

The VP method handles scenario S_1 but it cannot handle scenarios S_2 and S_3. In addition, Example 3 given in Section 3.4 shows a case where the VP method [42] provides a *false positive* result for a scenario involving code motion across loops. Moreover, the VP method does not check whether a computation is a false computation i.e., it never executes. As a result, it gives *false negative* results in the case of loop invariant code motion involving *false computations*.

3.1.3 Contributions

In this chapter, we present an equivalence checking method based on value propagation for code motion involving loops to overcome all the above limitations of existing works. Our method is capable of handling all the three scenarios, i.e., S_1, S_2 and S_3, mentioned above. Moreover, our method is able to prove non-equivalence for the case given in Example 3. Also, if the loop is executed at least once, then our method will ignore the false computation during equivalence checking. In particular, a bug in the HLS tool SPARK [15] involving loop invariant code motion is detected by our method.

The rest of this Chapter is organized as follows. The FSMD model and the VP method are explained in Sections 3.2 and 3.3, respectively. Motivating examples highlighting the limitations of the VP method are given in Section 3.4. A solution to handle all the above scenarios and to identify a false computation of an FSMD during equivalence checking is presented in Section 3.5. The enhanced VP method is presented in Section 3.6. Experimental results are given in Section 3.8. Section 3.9 concludes the chapter.

3.2 The FSMD Model

In the translation validation approach, the input behavior to HLS (i.e., source behavior) and the scheduled behavior generated at the scheduling phase of HLS (i.e., transformed behavior) are represented using the FSMD model. FSMDs [1] are an extension of the finite state machine (FSM) model with data/variables used to model behaviors. Unlike FSMs that model the control flow, FSMDs capture the data-flow aspect of the behavior as well. Each transition of an FSMD includes a condition over the data variables and a set of operations that transform the variable values.

Definition 1 (FSMD)**.** *An FSMD M is defined as a 7-tuple $\langle Q, q_0, I, O, V, f, h\rangle$, where*

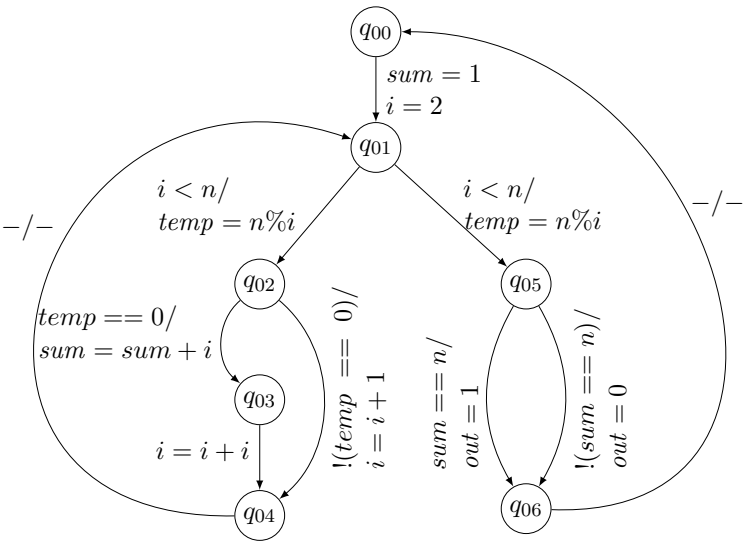
- *Q is the finite set of states,*
- *$q_0 \in Q$ is the reset (initial) state,*
- *I is the finite set of input variables,*
- *O is the finite set of output variables,*
- *V is the finite set of storage variables,*
- *$f : Q \times 2^S \rightarrow Q$ is the state transition function,*
- *$h : Q \times 2^S \rightarrow U$ is the update function.*

Here $S = \{L \cup E\}$ is the set of status expressions where L is the set of Boolean literals of the form b or $\neg b$, $b \in B \subseteq V$ is a Boolean variable and E is the set of arithmetic predicates over $I \cup (V - B)$. Any arithmetic predicate is of the form $eR0$, where e is an arithmetic expression and $R \in \{==, \neq, >, \geq, <, \leq\}$. U is a set of storage or output assignments of the form $\{x = e | x \in O \cup V\}$ and e is an arithmetic predicate or expression over $I \cup (V - B)$; it represents a set of storage or output assignments. An FSMD is an inherently deterministic model.

A *walk* from q_i to q_j is a sequence of state transitions of the form $\langle q_i \overset{c_i}{\Rightarrow} q_{i+1} \overset{c_{i+1}}{\Longrightarrow} \cdots \overset{c_{i+n-1}}{\Longrightarrow} q_{i+n} = q_j\rangle$ where $q_k \in Q \ \forall\ k$, $i \leq k \leq i + n$, and the state transitions $f(q_k, c_k) = q_{k+1}$ for all k, $i \leq k \leq i + n - 1$, where $c_k \in 2^S$ is the condition of

```
1  sum=1;
2  i=2;
3  while(i<n){
4    temp=n%i;
5    if(temp==0){
6      sum=sum+i;
7      i=i+1;}
8    else
9     i=i+1;}
10 if(sum==n)
11   out=1;
12 else
13   out=0;
```

(a) Input behavior



(b) FSMD M_0

Figure 3.3: An FSMD example: (a) finds whether a number n is a perfect number or not; (b) an FSMD M_0 corresponding to input behavior (a)

the transition from q_k to q_{k+1}. A *(finite) path* β is a walk where all the states are different, except the end state q_j may be the same as the start state q_i. The *condition of execution* R_β of a path β is a logical expression over $I \cup V$, which must be satisfied by the initial data state in order to traverse the path β. The *data transformation* (s_β) represents the updated variables vector $s_\beta = \langle e_1, e_2, \ldots, e_k \rangle$, where $k = |V|$ and e_i is an algebraic expressions over the variables in V and the inputs in I. The expression e_i represents the symbolic value obtained by the variable $v_i \in V$ at the end state of β when the initial symbolic value of the variable v is denoted as 'v'. For a path β, R_β and s_β are computed by forward or backward substitution based on symbolic execution [100].

It may be noted that there would be an ordered list of outputs in any path as discussed in [38]. For equivalence of two paths, the outputs of them also must match. When some variable is output, its counterpart in the other FSMD must attain the same value. Therefore, the equivalence of outputs hinges upon the equivalence of data transformations of variables (i.e., s_β). Hence, in this work, we focus only on equivalence of s_β. The paper [39] discusses in detail how the FSMD models can be constructed from the high-level representations of the input and

the transformed behaviors.

Example 1. *Let us consider the input behavior in Fig. 3.3(a) and its corresponding FSMDs in Fig. 3.3(b). The behavior checks whether a number n is perfect number or not. If n is a perfect number then it sets the value of the variable out to 1 otherwise 0. Let consider the path* $\beta = q_{00} \Rightarrow q_{01} \xrightarrow{!(i<n)} q_{05} \xrightarrow{sum==n} q_{06} \Rightarrow q_{00}$ *in the FSMD* M_0 *in Fig. 3.3(b). The computation of* $[R_\beta, s_\beta]$ *for this path* β *by forward substitution method is as follows:*
At q_{00} : $[\texttt{True}, \langle sum, i, n, temp, out\rangle]$.
At q_{01} : $[\texttt{True}, \langle 1, 2, n, temp, out\rangle]$.
At q_{05} : $[!(2 < \texttt{n}), \langle 1, 2, n, temp, out\rangle]$.
At q_{06} : $[!(2 < \texttt{n}) \wedge (2 == \texttt{n}), \langle 1, 2, n, temp, 1\rangle]$.
At q_{00} : $[!(2 < \texttt{n}) \wedge (2 == \texttt{n}), \langle 1, 2, n, temp, 1\rangle]$.

3.2.1 Equivalence of FSMDs

Let $M_0 = \langle Q_0, q_{00}, I, O, V_0, f_0, h_0\rangle$ and $M_1 = \langle Q_1, q_{10}, I, O, V_1, f_1, h_1\rangle$ be two FSMDs having the same input(s)/output(s). A computation of an FSMD is a finite walk from the reset state q_0 to itself, and q_0 should not occur in between. The M_1 is derived from M_0 through HLS scheduling. Our main goal is to verify whether M_0 behaves exactly as M_1. This means that for all possible input sequences, M_0 and M_1 produce the same sequences of output values and eventually, when the respective reset states are revisited, they are visited with the same storage element values. In other words, for every computation from the reset state back to itself of one FSMD, there exists an equivalent computation from the reset state back to itself in the other FSMD and vice versa.

Definition 2 (Computation Equivalence)**.** *Two computations* μ_0 *and* μ_1 *are equivalent, denoted as* $\mu_0 \simeq \mu_1$ *iff* $R_{\mu_0} \equiv R_{\mu_1}$ *and* $s_{\mu_0} = s_{\mu_1}$*, where* R_{μ_0} *and* R_{μ_1} *are the condition of execution of* μ_0 *and* μ_1*, respectively and* s_{μ_0} *and* s_{μ_1} *are the data transformation of* μ_0 *and* μ_1*, respectively.*

Definition 3 (FSMD Containment)**.** *An FSMD* M_0 *is contained in another FSMD* M_1 *(*$M_0 \sqsubseteq M_1$*) if for any computation* μ_0 *of* M_0 *on some inputs, there exists a computation* μ_1 *of* M_1 *on the same inputs such that* $\mu_0 \simeq \mu_1$*.*

Definition 4 (Equivalence of FSMDs)**.** *Two FSMDs M_0 and M_1 are computationally equivalent, if $M_0 \sqsubseteq M_1$ and $M_1 \sqsubseteq M_0$.*

An FSMD may consist of an infinite number of computations because of loops. However, for an FSMD M, any computation μ is the concatenation $[\beta_1\beta_2\cdots\beta_n]$ of paths of M where for all k, $1 \leq k < n$, β_k terminates in the start state of the path β_{k+1}; the reset state is the start state of β_1 and the end state of β_n; β_i's may not all be distinct. Hence, we have the following definition.

Definition 5 (Path cover of an FSMD)**.** *A finite set of paths $P = \{\beta_0, \beta_1, \ldots, \beta_k\}$ is said to be a path cover of an FSMD M if any computation μ of M can be looked upon as a concatenation of paths from P.*

To obtain a path cover for an FSMD model each loop is cut at at least one cutpoint. The set of all paths from a cutpoint to another cutpoint without any intermediary occurrence of a cutpoint is a path cover of the FSMD. This is based on the Floyd–Hoare method of program verification [54].

Definition 6 (Path Equivalence)**.** *Two paths β and α are equivalent denoted by $\beta \simeq \alpha$ if $R_\beta \equiv R_\alpha$ and $s_\beta = s_\alpha$.*

The correspondence of states between M_0 and M_1 are defined as follow.

Definition 7 (Corresponding States)**.**

1. *The reset states q_{00} and q_{10} are corresponding states.*
2. *The states $q_{0k} \in Q_0$ and $q_{1l} \in Q_1$ are corresponding states if the state $q_{0i} \in Q_0$ and $q_{1j} \in Q_1$ are corresponding states and there exists paths β from q_{0i} to q_{0k} and α from q_{1j} to q_{1l}, such that $\beta \simeq \alpha$.*

The following theorem can be concluded from the above discussion.

Theorem 1. *An FSMD M_0 is contained in another FSMD M_1 ($M_0 \sqsubseteq M_1$), if there exists a path cover $P_0 = \{\beta_{00}, \beta_{01}, \cdots, \beta_{0k}\}$ of M_0 and $P_1 = \{\alpha_{10}, \alpha_{11}, \cdots, \alpha_{1k}\}$ of M_1 such that $\beta_{0i} \simeq \alpha_{1i}$ for all i, $0 \leq i \leq k$.*

Proof. $M_0 \sqsubseteq M_1$ if, for any computation μ_0 of M_0, there exists a computation μ_1 of M_1 such that μ_0 and μ_1 are computationally equivalent. [by Definition 2]

Now, let there exists a path cover $P_0 = \{\beta_{00}, \beta_{01}, \cdots, \beta_{0k}\}$ of M_0. Corresponding to P_0, let a set $P_1 = \{\alpha_{10}, \alpha_{11}, \cdots, \alpha_{1k}\}$ of M_1 exists such that $\beta_{0i} \simeq \alpha_{1i}$ for all i, $0 \leq i \leq k$. Since P_0 covers M_0, any computation μ_0 of M_0 can be looked upon as a concatenated path $[\beta_{0i_1}, \beta_{0i_2}, \ldots, \beta_{0i_n}]$ from P_0 starting from the reset state (q_{00}) and ending again at the reset state of M_0. From above it follows that there exists a sequence Π_1 of paths $[\alpha_{1j_1}, \alpha_{1j_2}, \ldots, \alpha_{1j_n}]$ of P_1, where $\beta_{0i_l} \simeq \alpha_{1j_l}$ for all l, $0 \leq l \leq n$. Therefore, in order that Π_1 represents a computation of M_1, it is required to prove that Π_1 is a concatenated path of M_1 from its reset state q_{10} back to itself.

Now, let $\beta_{0i_1} : [q_{00} \Rightarrow q_{0f_1}]$. Since $\beta_{0i_1} \simeq \alpha_{1j_1}$, from the definition of corresponding states, α_{1j_1} must be of the form $[q_{10} \Rightarrow q_{1f_1}]$, where $\langle q_{00}, q_{10}\rangle$ and $\langle q_{0f_1}, q_{1f_1}\rangle$ are corresponding states. Thus, by repetitive application of the above argument, it follows that if $\beta_{0i_1} : [q_{00} \Rightarrow q_{0f_1}], \beta_{0i_2} : [q_{0f_1} \Rightarrow q_{0f_2}], \cdots, \beta_{0i_n} : [q_{0f_{n-1}} \Rightarrow q_{0f_n} = q_{00}]$, then $\alpha_{1j_1} : [q_{10} \Rightarrow q_{1f_1}], \alpha_{1j_2} : [q_{1f_1} \Rightarrow q_{1f_2}], \cdots, \alpha_{1j_n} : [q_{1f_{n-1}} \Rightarrow q_{1f_n} = q_{10}]$, where $\langle q_{0f_m}, q_{1f_m}\rangle$, $1 \leq m \leq n$, are pairs of corresponding states. Hence, Π_1 is a concatenated path representing a computation μ_1 of M_1, where $\mu_0 \simeq \mu_1$. ■

Two FSMDs M_0 and M_1 are equivalent, denoted as $M_0 \equiv M_1$, if $M_0 \sqsubseteq M_1$ and $M_1 \sqsubseteq M_0$. Since FSMDs are deterministic, it can be shown that $M_0 \sqsubseteq M_1$ implies $M_1 \sqsubseteq M_0$.

3.3 Value Propagation Based Equivalence of FSMDs

The value propagation method consists in propagating values of variables over the corresponding paths of two FSMDs on discovery of mismatch in the values of some variables. Propagation of values from a path β_1 to the subsequent path β_2 is carried out by associating a propagated vector at the end state of the path β_1 (or equivalently, the start state of the path β_2). A *propagated vector* ϑ_{β_f} at the end state q_{β_f} [1] of a path β is an ordered pair $\langle R'_{\beta_f}, s'_{\beta_f}\rangle$, where the first element is the condition of execution (R_β) and the second element is the vector of values of the variables of both FSMDs when the path β is compared with another path α in

[1]The start state and the final state of a path β is denoted as q_{β_s}, q_{β_f}, respectively.

the other FSMD. Let say that $\bar{v}$ denotes $\langle v_1, v_2 \cdots v_k \rangle$ and $\bar{e}$ denotes $\langle e_1, e_2 \cdots e_k \rangle$ where e_i is the symbolic expression involving variables in $\bar{v}$. The propagated vector associated with the reset state is $\langle \mathtt{T}, \langle v_1, v_2 \ldots v_k \rangle \rangle$, also denoted as $\bar{\rho}$, where `T` stands for `True` and $e_i = v_i, 1 \leq i \leq k$ indicates that the variables are yet to define.

Let there is a path $\beta : q_{\beta_s} \Rightarrow q_{\beta_f}$ in an FSMD M_0 with a propagated vector ϑ_{β_s} associated with q_{β_s} and a path $\alpha : q_{\alpha_s} \Rightarrow q_{\alpha_f}$ in an FSMD M_1 with a propagated vector ϑ_{α_s} associated with q_{α_s}. The *characteristic formula* for the path β is $\tau_\beta^{\vartheta_{\beta_s}} = \langle R_\beta^{\vartheta_{\beta_s}}, s_\beta^{\vartheta_{\beta_s}} \rangle$, where $R_\beta^{\vartheta_{\beta_s}}$ is the condition of execution of β and $s_\beta^{\vartheta_{\beta_s}}$ is the data transformation of β considering the data state of the variables at q_{β_s} is ϑ_{β_s} (instead of $\bar{v}$). Similarly, the characteristic formula for the path α is $\tau_\alpha^{\vartheta_{\alpha_s}} = \langle R_\alpha^{\vartheta_{\alpha_s}}, s_\alpha^{\vartheta_{\alpha_s}} \rangle$.

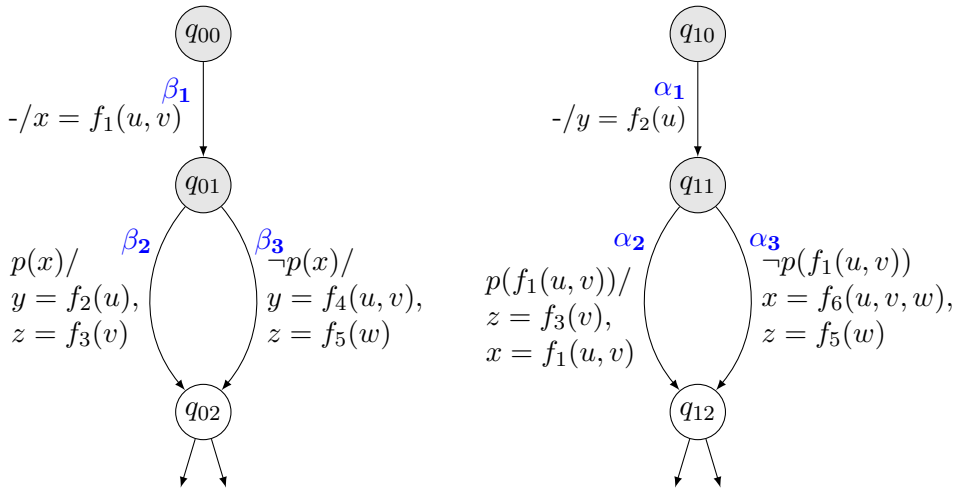
In the VP method a path cover is obtained by setting the reset state and the branching states (i.e., states with more than one outward transition) of the FSMD as cutpoints. To check the equivalence between two paths say β of FSMD M_0 and α of M_1, the characteristic formula associated with these path are compared.

Definition 8 (Unconditionally and Conditionally Equivalent Paths)**.** *A path $\beta : q_{\beta_s} \Rightarrow q_{\beta_f}$ with a characteristic formula $\tau_\beta^{\vartheta_{\beta_s}} = \langle R_\beta^{\vartheta_{\beta_s}}, s_\beta^{\vartheta_{\beta_s}} \rangle$ is said to be unconditionally equivalent (U-equivalent in short, denoted by $\beta \simeq_u \alpha$) if $R_\beta^{\vartheta_{\beta_s}} \equiv R_\alpha^{\vartheta_{\alpha_s}}$ and $s_\beta^{\vartheta_{\beta_s}} = s_\alpha^{\vartheta_{\alpha_s}}$. Otherwise, the path β is said to be conditionally equivalent (C-equivalent in short, denoted by $\beta \simeq_c \alpha$) if*

- *$q_{\beta_f} \neq q_{00}$ and $q_\alpha \neq q_{10}$.*
- *$\forall \beta'$ emanating from the state q_{β_f} with propagated vector $\langle R'_{\beta_f}, s'_{\beta_f} \rangle$ there exists a path α' emanating from q_{α_f} with the propagated vector $\langle R'_{\alpha_f}, s'_{\alpha_f} \rangle$, such that $\beta' \simeq_u \alpha'$ or $\beta' \simeq_c \alpha'$.*

Once a C-equivalent path is identified, the VP method tries to find a U-equivalent path in a depth-first search (DFS) manner. Example 2 illustrates the method of value propagation.

Example 2. *Let us consider the input FSMD in Fig. 3.4(a) and the transformed FSMD in Fig. 3.4(b). Let the variable ordering be $\langle u, v, w, x, y, z \rangle$. The propagated vector at the reset state q_{00} (q_{10}) is $\vartheta_{00}(\vartheta_{10}) = \langle \mathtt{T}, \langle u, v, w, x, y, z \rangle \rangle$. The characteristic formula for the path β_1 is $\tau_\beta^{\vartheta_{00}} = \langle \mathtt{T}, \langle u, v, w, f_1(u, v), y, z \rangle \rangle$ and for the*



(a) M_0 (b) M_1

Figure 3.4: An example of value propagation

path α *is* $\tau_\alpha^{\vartheta_{10}} = \langle \mathtt{T}, \langle u, v, w, x, f_2(u), z\rangle\rangle$. *In* $\tau_\beta^{\vartheta_{00}}$ *and* $\tau_\alpha^{\vartheta_{10}}$, *there is a mismatch in the values of* x *and* y. *Therefore, the propagated vector at* q_{01} *and* q_{11} *are* $\vartheta_{01} = \langle \mathtt{T}, \langle u, v, w, \mathbf{f_1}(\mathbf{u}, \mathbf{v}), \mathbf{y}, z\rangle\rangle$ *and* $\vartheta_{11} = \langle \mathtt{T}, \langle u, v, w, \mathbf{x}, \mathbf{f_2}(\mathbf{u}), z\rangle\rangle$ *respectively. In* ϑ_{01} *and* ϑ_{11} *the values of* x *and* y *are in boldface to denote that they mismatch and other variables whose value matches are in the normal face. The characteristic formula for* β_2 *with respect to* ϑ_{01} *is* $\tau_{\beta_2}^{\vartheta_{01}} = \langle \mathtt{T} \wedge p(f_1(u, v)), \langle u, v, w, f_1(u, v), f_2(u), f_3(v)\rangle\rangle$ *and for* α_2 *with respect to* ϑ_{11} *is* $\tau_{\beta_2}^{\vartheta_{11}} = \langle \mathtt{T} \wedge p(f_1(u, v)), \langle u, v, w, f_1(u, v), f_2(u), f_3(v)\rangle\rangle$. *The characteristic formulas* $\tau_{\beta_2}^{\vartheta_{01}}$ *and* $\tau_{\alpha_2}^{\vartheta_{11}}$ *are equal therefore the propagated vector at* q_{02} *and* q_{12} *are* $\bar{\rho}$. *The characteristic formula for* β_3 *with respect to* ϑ_{01} *is* $\tau_{\beta_3}^{\vartheta_{01}} = \langle \mathtt{T} \wedge \neg p(f_1(u, v)), \langle u, v, w, f_1(u, v), f_4(u, v), f_5(w)\rangle\rangle$ *and for* α_3 *with respect to* ϑ_{11} *is* $\tau_{\alpha_3}^{\vartheta_{11}} = \langle \mathtt{T} \wedge \neg p(f_1(u, v), \langle u, v, w, f_6(u, v, w), f_2(u), f_5(w)\rangle\rangle$. *There is a mismatch in therefore the propagated vector at (via* β_3*)* q_{02} *and (via* α_3*)* q_{12} *are* $\vartheta_{02} = \langle \mathtt{T} \wedge \neg p(f_1(u, v)), \langle u, v, w, \mathbf{f_1}(\mathbf{u}, \mathbf{v}), \mathbf{f_4}(\mathbf{u}, \mathbf{v}), f_5(w)\rangle\rangle$ *and* $\vartheta_{12} = \langle \mathtt{T} \wedge \neg p(f_1(u, v), \langle u, v, w, \mathbf{f_6}(\mathbf{u}, \mathbf{v}, \mathbf{w}), \mathbf{f_2}(\mathbf{u}), f_5(w)\rangle\rangle$. *It may be noted the variable* z *is not reverted to its symbolic value even though it matches in both of the propagated vectors.*

An abstract version of the VP method is given in Algorithm 1. The details can be found in [42]. The function `containmentChecker` (Algorithm 1) identifies the cutpoints and a path cover in an FSMD. It invokes `correspondenceChecker` (Al-

Algorithm 1: `containmentChecker`(FSMD M_0, FSMD M_1)

```
1  M0 and M1 and compute their path cover P0 and P1; Wcsp is a set of
     corresponding state pairs and initially contains (q00, q10);
2  foreach (q0i, q1j) ∈ Wcsp do
3  |  if correspondenceChecker (q0i, q1j, P0, P1, Wcsp) returns "failure" then
4  |  |  Report "unable to decide M0 ⊑ M1" and exit;
5  |  end if
6  end foreach
7  Report "M0 ⊑ M1";
```

Algorithm 2: `correspondenceChecker`$(q_{0i}, q_{1j}, P_0, P_1, W_{csp})$

```
1  foreach path β : (q0i ⇒ q0m) in P0 do
2  |  if path α : (q1j ⇒ q1n) can be found in P1 such that β ≃u α then
3  |  |  Wcsp = Wcsp ∪ {(q0m, q1n)};
4  |  else if path α : (q1j ⇒ q1n) can be found in P1 such that β ≃c α then
5  |  |  if q0m or q1n is the reset state then
6  |  |  |  return failure;
7  |  |  else
8  |  |  |  correspondenceChecker(q0m, q1n, P0, P1, Wcsp);
9  |  |  end if
10 |  else
11 |  |  return failure;
12 |  end if
13 end foreach
14 return success;
```

gorithm 2) for each corresponding state pairs, one by one. The `correspondenceChecker` function checks whether for every path emanating from a state in the pair, there is a U- or C-equivalent path from the other member of the pair. Based on the output returned by `correspondenceChecker`, `containmentChecker` reports whether the source FSMD is contained in the transformed FSMD or not.

3.4 Motivations

In this section, we present a case where the VP method provides a false positive result. We also present a case where the VP method reports a possible non-equivalence of FSMDs which are actually equivalent due to the presence of a false

computation.

3.4.1 False Positive Case of the VP Method

To detect valid code motion across a loop, the VP method marks the variables which exhibit a mismatch in the propagated vector. Those variables on which these marked variables depend are also marked in the propagated vector. The rest of the variables are denoted as unmarked variables. A code motion across a loop is determined to be valid by the VP method iff

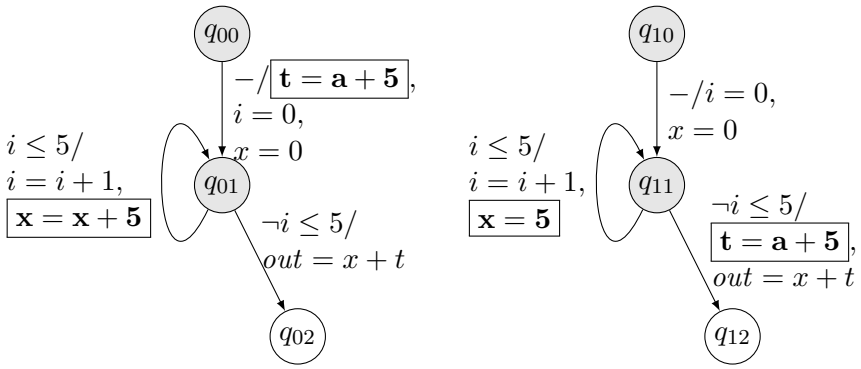
1. the values of marked variables are exactly the same after exiting the loop as before entering the loop in both behaviors and

2. the data transformations of unmarked variables, with respect to the propagated vector (stored before entering the loop) are exactly the same within the loop in each behavior.

In other words, the marked variables should not be updated within the loop in each behavior, and the unmarked variables should be updated in exactly the same manner in both behaviors. It may be noted that after traversing the loop once, the VP method compares the unmarked variable values of each behavior. If the values are the same, then it declares that all the variables are identically defined. But this may not always be true as shown in Example 3. Therefore, the VP method produces *false positive* results. In a propagated vector, we use bold face to denote the marked variables. Example 3 illustrates a scenario where two behaviors are not equivalent but this method declares them equivalent.

Example 3. *In Fig. 3.5, the operation* $t = a+5$ *is moved across the loop as shown in FSMDs* M_0 *and* M_1*. Let the variable ordering be* $\langle a, i, out, x, t\rangle$*. The operation* $x = x + 5$ *is intentionally replaced by* $x = 5$ *in* M_1*; clearly these two behaviors are not equivalent.*

The propagated vector at the reset state q_{00} (q_{10}) *is* $\vartheta_{00}(\vartheta_{10}) = \langle \mathtt{T}, \langle a, i, out, x, t\rangle\rangle$*. For the path* $q_{00} \Rightarrow q_{01}$ *of* M_0*, the VP method finds the candidate C-equivalent path* $q_{10} \Rightarrow q_{11}$ *of* M_1*. So the propagated vectors at* q_{01} *and* q_{11} *are* $\vartheta_{01} = \langle \mathtt{T}, \langle \mathbf{a}, 0, out, 0, \mathbf{a+5}\rangle\rangle$ *and* $\vartheta_{11} = \langle \mathtt{T}, \langle \mathbf{a}, 0, out, 0, \mathbf{t}\rangle\rangle$*, respectively. The VP method checks all the paths emanating from the state* q_{01} *and its corresponding state* q_{11}



(a) M_0 (b) M_1

Figure 3.5: An example where the VP method gives false positive result.

with respect to propagated vector ϑ_{01} and ϑ_{11}, respectively. It may be noted that before entering the loop the propagated vectors at q_{01} and q_{11} are not the same because of the mismatch in the value of variable t. In this case t and a are marked variables because the values of t do not match and t depends on a. After traversing the loop once, the propagated vectors at q_{01} and q_{11} will be $\vartheta'_{01} = \langle T, \langle \mathbf{a}, 1, out, 5, \mathbf{a} + \mathbf{5} \rangle \rangle$ and $\vartheta'_{11} = \langle T, \langle \mathbf{a}, 1, out, 5, \mathbf{t} \rangle \rangle$ respectively. Here the marked variables t and a are not updated in either of the loops (i.e., condition 1 is satisfied) and the unmarked variables x and i have the same transformation (the value of x is 5 and the value of i is 1) in both the loops (i.e., thus satisfy the condition 2) with respect to propagated vectors ϑ_{01} and ϑ_{11}. Therefore, the VP method says it is a valid case of code motion across a loop. Since it cannot be determined statically how many times a loop will execute, all the unmarked variable are reverted to their symbolic value at the exit of the loop in the VP method. Therefore, the propagated vector at q_{01} and q_{11} at the end of the loop will be $\vartheta'_{01} = \langle T, \langle \mathbf{a}, i, out, x, \mathbf{a} + \mathbf{5} \rangle \rangle$ and $\vartheta'_{11} = \langle T, \langle \mathbf{a}, i, out, x, \mathbf{t} \rangle \rangle$ respectively. For the path $q_{01} \Rightarrow q_{02}$ of M_0 and for the path $q_{11} \Rightarrow q_{12}$, the propagated vector at q_{02} and q_{12} will be $\vartheta_{02} = \langle \mathtt{T}, \langle a, i, x + t, x, t + 5 \rangle \rangle$ and $\vartheta_{12} = \langle \mathtt{T}, \langle a, i, x + t, x, t + 5 \rangle \rangle$ respectively. The propagated vector ϑ_{02} and ϑ_{12} are the same at q_{02} and q_{12}. Finally, $q_{01} \Rightarrow q_{02}$ and $q_{11} \Rightarrow q_{12}$ are designated as a U-equivalent, and the previously declared candidate C-equivalent path pairs are asserted to be C-equivalent. Hence, the VP method declares $M_0 \equiv M_1$.

It may be noted that after exiting the loop the value of x at q_{01} will be 30 in M_0; while, it will be the value 5 at q_{11} in M_1. Clearly, these two behaviors are not

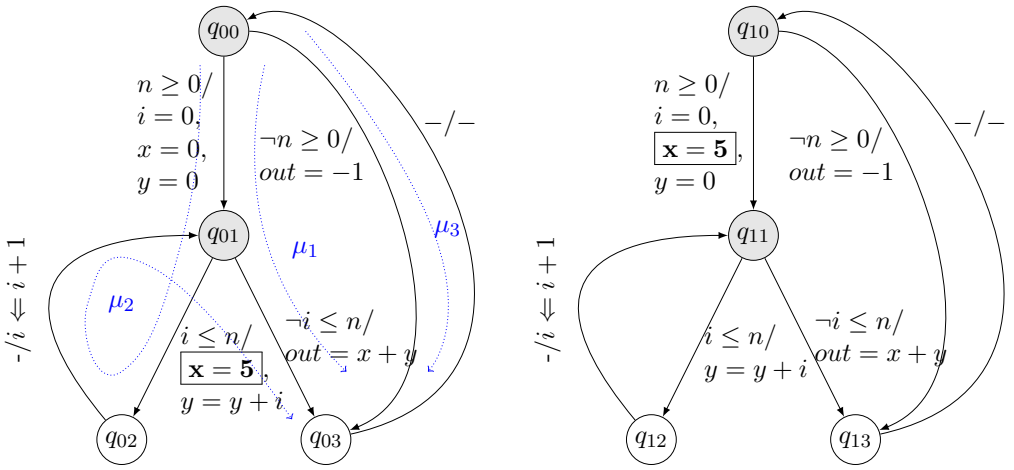


Figure 3.6: An example where the VP method provides false negative result.

equivalent. Hence, the VP method gives a false positive result in this case.

In the case of mismatch at the loop header (i.e., entry point of the loop), the VP method does not revert all the unmarked variables to their symbolic values and propagates their values along with the marked variables. It may cause the VP method to produce a false positive result in some scenario. In Example 3 the VP method not able to detect the mismatch for unmarked variable x at the of the loop. To avoid the false positive result the VP method should propagate only the marked variable values and all the unmarked variables should be reverted to their symbolic values. In the Subsection 3.5.1 we propose a solution to show the non-equivalence for false positive cases.

3.4.2 False Computation Involving Loops

Example 4 illustrates a case where the VP method provides false negative results due to the presence of a false computation.

Example 4. *Let us consider the FSMDs in Fig. 3.6. In this example, the operation* $x = 5$ *is a loop invariant for FSMD* M_0 *in Fig. 3.6(a). It is placed before the loop in the transformed FSMD* M_1 *in Fig. 3.6(b).*

There are three possible computations, $\mu_1 = \langle q_{00} \overset{n\geq 0}{\Longrightarrow} q_{01} \overset{\neg i\leq n}{\Longrightarrow} q_{03} \Rightarrow q_{00}\rangle$, $\mu_2 = \langle q_{00} \overset{n\geq 0}{\Longrightarrow} (q_{01} \overset{i\leq n}{\Longrightarrow} q_{01})^+ \overset{\neg i\leq n}{\Longrightarrow} q_{03} \Rightarrow q_{00}\rangle$ *and* $\mu_3 = \langle q_{00} \overset{\neg n\geq 0}{\Longrightarrow} q_{03} \Rightarrow q_{00}\rangle$ *for the FSMD in Fig. 3.6(a). The computation* μ_1 *executes if the loop condition* $i \leq n$ *is* `False` *for* $n \geq 0$. *The computation* μ_2 *executes if the loop condition* $i \leq n$ *is* `True` *for the input* $n \geq 0$. *The computation* μ_3 *executes if* $n < 0$ *holds. In this example, when the state* q_{01} *is reached for the first time,* n *is always greater than or equal to 0, and* i *is equal to 0. Therefore, the computation* μ_1 *will never execute. In other words, the loop will execute at least once for all possible* $n \geq 0$ *and* $i = 0$. *The computation* μ_1 *is, therefore, a false computation.*

The VP method explores all possible computations of a given FSMD M_0. *It does not check whether a computation is a false computation or not. It finds that the computation* μ_2 *and* μ_3 *of FSMD* M_0 *are equivalent to the computation* $\langle q_{10} \overset{n\geq 0}{\Longrightarrow} (q_{11} \overset{i\leq n}{\Longrightarrow} q_{11})^+ \overset{\neg i\leq n}{\Longrightarrow} q_{13} \Rightarrow q_{10}\rangle$, $\langle q_{10} \overset{\neg n\geq 0}{\Longrightarrow} q_{13} \Rightarrow q_{10}\rangle$ *of FSMD* M_1, *respectively. However, the VP method fails to find* $\langle q_{10} \overset{n\geq 0}{\Longrightarrow} q_{11} \overset{\neg i\leq n}{\Longrightarrow} q_{13} \Rightarrow q_{10}\rangle$ *as an equivalent computation of* μ_1 *in FSMD* M_0, *since they differ in the final value of the variable* x. *It may be noted that the final value of* x *would be 0 after execution of* μ_1 *in* M_0 *and 5 after the execution of* $\langle q_{10} \overset{n\geq 0}{\Longrightarrow} q_{11} \overset{\neg i\leq n}{\Longrightarrow} q_{13} \Rightarrow q_{10}\rangle$ *in* M_1. *In this example, as described above, the computation* μ_1 *will never execute. The non-equivalence of FSMDs reported by the VP method is due to this false computation.*

If we can ignore this false computation during equivalence checking, we can establish the equivalence between these two behaviors. In Subsection 3.5.2 we show how to ignore a false computation with the help of SMT solver Z3 [57].

3.4.3 Code Motion Involving Loops

The VP method is presented for validating code motion across the loop i.e., S_1 during translation. However, the VP Method cannot handle the scenario where some code segment is moved before (after) the loop from inside the loop body i.e., S_2 and S_3. In Subsection 3.5.3 we propose an enhancement to overcome this issue.

3.5 Proposed Solutions

In this section, we propose a solution to prove the non-equivalence for the case given in Example 3. We also propose a solution to identify a false computation in an FSMD during equivalence checking. We also propose a solution to handle all the scenarios S_1, S_2 and S_3 during equivalence checking.

3.5.1 Showing the Non-Equivalence for False Positive Cases

The VP method propagates the values (as a propagated vector) of live variables over the corresponding paths of the two behaviors as follows.

- If there is a mismatch in the propagated vector in a corresponding state pair, then it propagates not only the mismatched values (corresponding to marked variables), but also the matched values (corresponding to unmarked variables) as well. This is shown in Example 2.
- If there is no mismatch in the propagated vector in a corresponding state pair then all variables are reverted back to their symbolic values.

Our equivalence checking method is based on propagating the mismatch values of the variables through all the subsequent path segments until the values match or the final path segment ending in the reset state is reached. *However, in case of a mismatch at the loop header, we propagate only the marked variable values and all the unmarked variables will be reverted to their symbolic values.* This will help us to identify whether an unmarked variable is defined identically in both the loops. To ensure the validity of code motion like the VP method, we must ensure that marked variable should not be modified inside the loop body. The VP method fails to do this since in case of mismatch it propagates the actual value of variables and thus gives *false positive* results as shown in Example 3. Using our rule, the propagated vector at q_{01} (via $q_{00} \Rightarrow q_{01}$ path) will be $\vartheta_{01} = \langle \mathtt{T}, \langle \mathbf{a}, i, out, x, \mathbf{a} + \mathbf{5} \rangle \rangle$ and the propagated vector q_{11} will be $\vartheta_{01} = \langle \mathtt{T}, \langle \mathbf{a}, i, out, x, \mathbf{t} \rangle \rangle$ (via $q_{10} \Rightarrow q_{11}$ path) before entering the loop in Example 3. At the end of the loop the propagated vector at q_{01} will be $\vartheta'_{01} = \langle \mathtt{i} \leq \mathtt{5}, \langle \mathbf{a}, i, out, x + 5, \mathbf{a} + \mathbf{5} \rangle \rangle$, and the propagated vector at q_{11} will be $\vartheta'_{11} = \langle \mathtt{i} \leq \mathtt{5}, \langle \mathbf{a}, i, out, 5, \mathbf{t} \rangle \rangle$. The value for x (unmarked variable) is not the same in ϑ'_{01} and ϑ'_{11}. Hence, it is not a valid code motion and the two behaviors shown in Fig. 3.5 are not equivalent.

```
for(i1 = L1; i1 ≤ H1; i1+ = r1)
  for(i2 = L2; i2 ≤ H2; i2+ = r2)
   ⋮
    for(in = Ln; in ≤ Hn; in+ = rn)
        Sn :   ...
```

Figure 3.7: Nested loop structure

3.5.2 Handling False Computation Involving Loops

Let us consider the generalized nested loop structure of depth n as shown in Fig. 3.7 for this purpose. Each iterator i_x, $1 \leq x \leq n$, is initialized to L_x. Each iterator i_x reaches its upper limit H_x by incrementing a step constant r_x.

The terms L_x and H_x, $x = 1, \ldots, n$, are assumed to be linear expressions over the input variables, constants or previous loop iterators $i_1 \cdots i_{x-1}$. These requirement on L_i, H_i, r_i and increment statement restrict the kind of loop to which our method will apply. Let us assume that C_p is a propagated condition at the start of the nested loop structure. Conceptually, the propagated condition in a state s is the condition of a path from the reset state of the behavior to the state s. In Fig. 3.6, for example, the C_p is $n \geq 0$ at state q_{01}. Under the condition C_p, the initial value of the loop iterator ($i_1 = L_1$) must satisfy the initial loop condition (i.e., $L_1 \leq H_1$) to execute a nested loop structure at least once. We can specify this condition by the following formula 3.1. If formula 3.1 is valid then a nested loop structure with nesting depth one will always execute at least once.

$$C_p \implies L_1 \leq H_1 \tag{3.1}$$

The formula 3.2 is the generalized form of the formula 3.1. If formula 3.2 is valid then the statement S_n at the generalized loop structure of nesting depth n, will

always execute at least once.

$$C_p \implies \Bigg(\exists i_1, \exists i_2, \cdots, \exists i_{n-1}, \exists a_1, \exists a_2, \cdots, \exists a_{n-1} \\ \Big(\big(L_n \leq H_n \big) \wedge \Big(\bigwedge_{x=1}^{n-1} f_x \Big) \Big) \Bigg) \tag{3.2}$$

where $f_x = \Big((L_x \leq i_x \leq H_x) \wedge (i_x = a_x r_x + L_x) \wedge (a_x \geq 0) \Big)$. The C_p is the propagated condition before entering the nested loop of depth n. We use these formulas to identify a false computation during equivalence checking. For checking the validity of these formulas, we use the SMT solver Z3 [57] in the theory of linear integer arithmetic. These formulas are dynamically generated in our equivalence checking framework.

For example, in Fig. 3.6 there is a loop $q_{01} \overset{i \leq n}{\Longrightarrow} q_{01}$ of nesting depth 1. At state q_{01} of FSMD M_0, the propagation condition C_p is $n \geq 0$. To verify whether the loop $q_{01} \overset{i \leq n}{\Longrightarrow} q_{01}$ will execute at least once, we should check the validity of the formula $n \geq 0 \implies 0 \leq n$. This formula is valid. Thus, the loop will always execute at least once for all possible values of $n \geq 0$. We can say that the computation $\langle q_{00} \overset{n \geq 0}{\Longrightarrow} q_{01} \overset{\neg i \leq n}{\Longrightarrow} q_{03} \Rightarrow q_{00} \rangle$ is a false computation. During equivalence checking, our method will ignore this false computation. By ignoring this false computation, we can show the equivalence between the two behaviors shown in Fig. 3.6.

3.5.3 Handling Loop Invariant Code Motion

We handle marked and unmarked variables separately at the loop header to handle the scenarios S_2 and S_3. Let q_{0i} be the entry/exit state of a loop body in M_0 and its corresponding state q_{1j} be the entry/exit state of a loop body in M_1. The state q_{0i} has the propagated vector ϑ_{0i} before entering the loop and the propagated vector ϑ'_{0i} after traversal of one of the path inside loop leading to q_{0i}. Similarly, state q_{1j} has the propagated vector ϑ_{1j} before entering the loop and the propagated vector ϑ'_{1j} after traversal of one the path inside loop leading to q_{1j}. During code motion involving loops following cases will arise:

Case 1 *Unmarked Variable*: There are two possibilities for an unmarked variable,

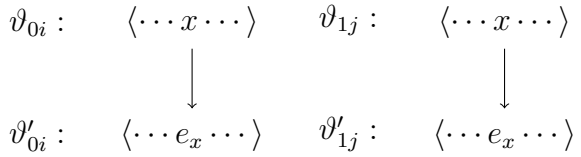


Figure 3.8: A case 1.1 where unmarked variable x is defined identically in both the loops

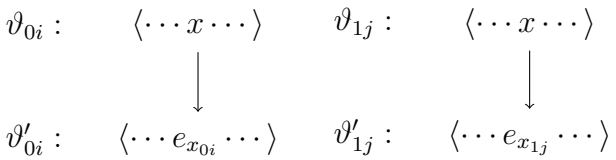


Figure 3.9: A case 1.2 where unmarked variable x has some mismatch at the end of the loop

say x. It may be noted that x has symbolic values in both ϑ_{0i} and ϑ_{1j}.

Case 1.1: If x has the same value in ϑ'_{0i} and ϑ'_{1j} then it indicates that x is defined identically in both the loops as shown in Fig. 3.8. Since it is not possible to determine statically how many times a loop will execute before exiting in this case, after exiting the loop x is reverted to its symbolic value.

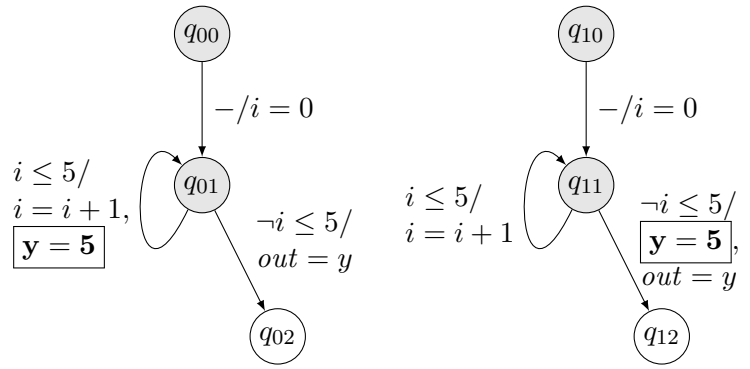
Case 1.2: If there is a mismatch for x in ϑ'_{0i} and ϑ'_{0j} then there is a possibility of the scenario S_3. Let $e_{x_{0i}}$ and $e_{x_{1j}}$ represent the mismatched values in ϑ'_{0i} and ϑ'_{1j} respectively as shown in Fig. 3.9. To check the validity of the code motion, we do the following test.

(a) The expressions $e_{x_{0i}}$ and $e_{x_{1j}}$ should be invariant in their corresponding loops.

(b) The variable x is not used before being defined in both the loops.

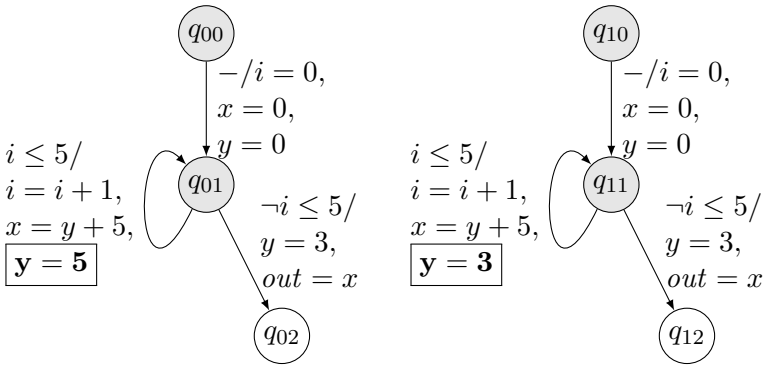
Examples 5 and 6 illustrate these cases.

Example 5. *Consider the input behavior M_0 and its transformed behavior M_1 in Fig. 3.10. The operation $y \Leftarrow 5$, a loop invariant for input behavior M_0, is placed after the loop body in the transformed behavior M_1. The input behavior*



(a) M_0 (b) M_1

Figure 3.10: An example of code motion involving scenarios S_3



(a) M_0 (b) M_1

Figure 3.11: An example where unmarked is used before being defined.

M_0 and the transformed behavior M_1, shown in Fig. 3.10, are equivalent. In both behaviors, ϑ_{00} and ϑ_{10} are $\langle \mathtt{T}, \langle i, out, y \rangle \rangle$ when entering the loop. After executing the loop once the propagated vector at q_{01} is $\vartheta'_{01} = \langle i \leq 5, \langle i + 1, out, \mathbf{5} \rangle \rangle$ and the propagated vector at q_{11} is $\vartheta'_{11} = \langle i \leq 5, \langle i + 1, out, \mathbf{y} \rangle \rangle$. The propagated vectors ϑ'_{01} and ϑ'_{11} differ in the value of y. Since, $y \Leftarrow 5$ is a loop invariant for loop body $(q_{01} \Rightarrow q_{01})$ and y is not used before defining it, it is a valid code motion. With the propagated vector ϑ'_{01} and ϑ'_{11}, the paths $q_{01} \Rightarrow q_{02}$ and $q_{11} \Rightarrow q_{12}$ can be shown to be equivalent.

Example 6. *Consider the input behavior M_0 and its transformed behavior M_1 in Fig. 3.11. These two behaviors shown in Fig. 3.11 are not actually equivalent since the values of x do not match after exiting the loop when the loop executes at*

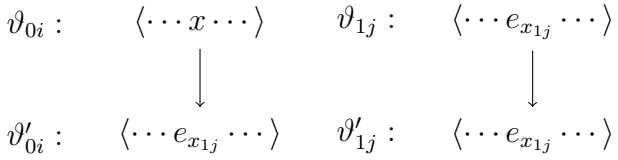


Figure 3.12: A case 2.1 where a marked variable x has the same value at the end of the loop

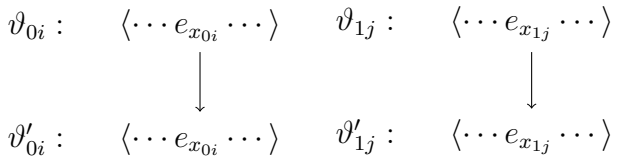


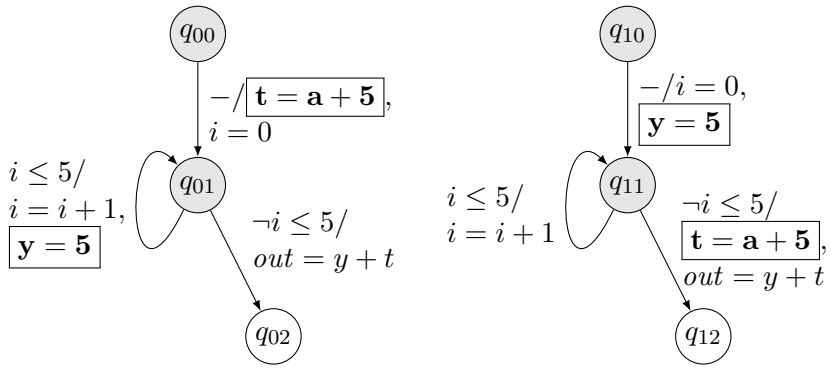
Figure 3.13: A case 2.3 where the values of the marked variable x do not update in both the loops

least two times. It may be noted that before entering the loop the variable x and y are unmarked variables at q_{01} and q_{11}. The variable x is defined identically in both the loops. The definition of y is a loop invariant in both the loops. However, the variable y is used in the definition of x before being defined. This will result in different values of x in the two behaviors after exiting the loop. Therefore, it is clear that we need to check both (1) if there is a mismatch for an unmarked variable then the mismatch should remain the same over multiple iterations and (2) for such a variable it has not been used before its definition in the loop in both the behaviors.

Case 2 *Marked Variable*: Marked variables arise in the case of S_1 and S_2. Here some definition before a loop is moved into the loop or is moved across the loop. Therefore, the marked variables may have some mismatch in the corresponding propagated vectors ϑ_{0i} and ϑ_{1j}.

Case 2.1: Suppose a marked variable, say x, has its symbolic value at ϑ_{0i} and $e_{x_{1j}}$ at ϑ_{1j}. If after executing the loop once the value of x matches in both the loops (i.e. x has the same value ($e_{x_{1j}}$) in ϑ'_{0i} and ϑ'_{1j}) as shown in Fig. 3.12, then scenario S_2 is possible. To check the validity of the code motion, we do the following test.

(a) The expression $e_{x_{1j}}$ should be invariant in both the loops.



(a) M_0 (b) M_1

Figure 3.14: An example of code motion involving scenarios S_1 and S_2

(b) The variable x is not used before being defined in the loop at q_{0i}, and it has no definition in the loop at q_{1j}.

Case 2.2: Suppose x has its symbolic value at ϑ_{1j} and $e_{x_{0i}}$ at ϑ_{0i} and after executing the loop once the value of x matches in both the loops. This case can be handled in a manner similar to case 2.1. However, this scenario unlikely to occurs in synthesis tools in practice.

Case 2.3: In the remaining case, if before executing the loop and after exiting the loop the value of x remains the same in both the loops as shown in Fig. 3.13 then scenario S_1 is possible. To check the validity of code motion, we do the following test.

(a) Variable x is not updated within the loop.

(b) All those variables on which the variable x depends should not be updated within the loop.

If a variable (x or any of the variables on which x depends) is updated within the loop (even identically for both FSMDs), then it is not a valid case of code motion. Example 7 illustrates this case.

Example 7. *Consider the input behavior M_0 and its transformed behavior M_0 in Fig. 3.14. The operation $y \Leftarrow 5$, a loop invariant for input behavior M_0, is placed out of the loop in the transformed behavior M_1. The operation $t \Leftarrow a + 5$ is*

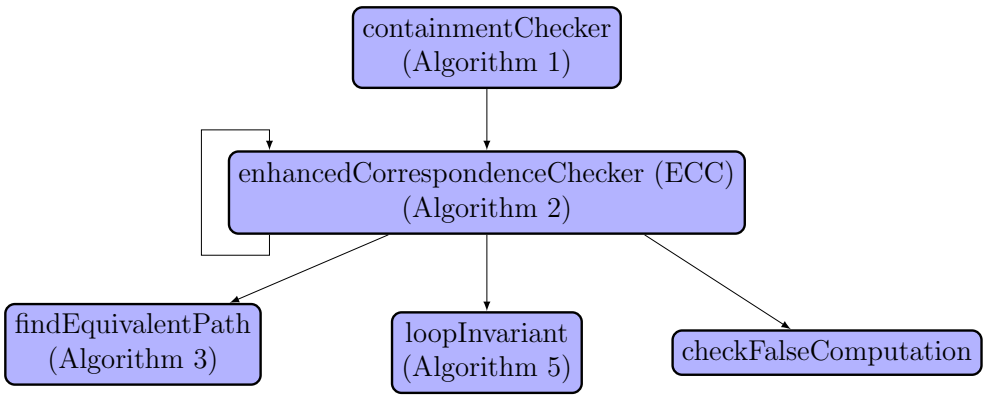


Figure 3.15: A overall flow of the EVP method

moved after the loop body. The input behavior M_0 and the transformed behavior M_1 shown in Fig. 3.14 are equivalent. It may be noted that before entering the loop the variables y, t and a are marked variables at states q_{01} and q_{11}. After exiting the loop the variable y has the same value at q_{01} and q_{11}. This is an instance of scenario S_2. The operation $y \Leftarrow 5$ is a loop invariant for both the loops and y is not defined within the loop at q_{11} hence this is valid code motion. After exiting the loop t and a remain the same as before entering the loop. This is an instance of scenario S_1. All the variables on which t and a depend are not updated in both the loops. Therefore this is also a valid code motion.

It may be noted that for the FSMDs M_0 shown in Figs. 3.10, 3.11, and 3.14 the loop will execute at least once. We can say that the computation $\langle q_{00} \Rightarrow q_{01} \overset{\neg i \leq 5}{\Longrightarrow} q_{02} \rangle$ is a false computation in each FSMD. So during equivalence checking our method will ignore this false computation.

3.6 Enhanced Value Propagation Based Equivalence Checking (EVP)

In this section, we present our enhanced VP method (EVP). We use the same *containmentChecker* fucntion of the VP method. We have enhanced the correspondence checker so that our method can handle all the issues address in Section 3.4. The `loopInvariant` function is also enhanced to handle all the cases discussed in Section 3.5.3.

Algorithm 3: `findEquivalentPath`$(\beta, \vartheta_{\beta_s}, q_{1j}, \vartheta_{q_{1j}}, P_0, P_1)$

Input : A path $\beta \in P_0$, the propagated vector ϑ_{β_s}, a state $q_{1j} \in M_1$, the propagated vector $\vartheta_{q_{1j}}$ associated with q_{1j}, and the path covers P_0 and P_1 of M_0 and M_1, respectively.

Output: An ordered tuple $\langle \gamma_1, \gamma_2, \vartheta_{\gamma_{1_f}}, \vartheta_{\gamma_{2_f}} \rangle$ s.t. $\gamma_1 \simeq_u \gamma_2$ or $\gamma_1 \simeq_c \gamma_2$, the propagated vectors $\vartheta_{\gamma_{1_f}}$ and $\vartheta_{\gamma_{2_f}}$

1 $\tau_\beta^{\vartheta_{\beta_s}} = \langle R_\beta^{\vartheta_{\beta_s}}, s_\beta^{\vartheta_{\beta_s}} \rangle$
2 **foreach** path $\alpha : (q_{\alpha_s} \Rightarrow q_{\alpha_f}) \in P_1$ emanating from q_{1j} `/* ` $q_{\alpha_s} = q_{1j}$ `*/`
3 **do**
4 $\quad \tau_\alpha^{\vartheta_{\alpha_s}} = \langle R_\alpha^{\vartheta_{\alpha_s}}, s_\alpha^{\vartheta_{\alpha_s}} \rangle$
5 $\quad$ **if** $R_\beta^{\vartheta_{\beta_s}} \equiv R_\alpha^{\vartheta_{\alpha_s}}$ **then**
6 $\quad\quad$ **if** $s_\beta^{\vartheta_{\beta_s}} = s_\alpha^{\vartheta_{\alpha_s}}$ **then**
7 $\quad\quad\quad$ **return** $(\beta, \alpha, \bar{\rho}, \bar{\rho})$;
8 $\quad\quad$ **else**
9 $\quad\quad\quad$ **return** $(\beta, \alpha, \vartheta_{\beta_f}, \vartheta_{\alpha_f})$;
10 $\quad\quad$ **end if**
11 $\quad$ **else if** $R_\alpha^{\vartheta_{\alpha_s}} \implies R_\beta^{\vartheta_{\beta_s}}$ **then**
12 $\quad\quad$ **return** $(\beta, \eta_{\alpha_s}, \vartheta_{\beta_f}, \vartheta_{\alpha_s})$;
13 $\quad$ **else if** $R_\beta^{\vartheta_\beta} \implies R_\alpha^{\vartheta_{\alpha_s}}$; **then**
14 $\quad\quad$ **return** $(\eta_{\beta_s}, \alpha, \vartheta_{\beta_s}, \vartheta_{\alpha_f})$;
15 $\quad$ **else**
16 $\quad\quad$ **continue**; `/* Case 6 --` $\beta \not\simeq \alpha$ `*/`
17 $\quad$ **end if**
18 **end foreach**
`/* All the paths emanating from` q_{1j} `are not equivalent to the path` β `*/`
19 **return** $(\beta, NULL, \bar{\rho}, \bar{\rho})$;

The overall flow of our verification method is given in Fig.3.15. The behavior of the enhanced correspondence checker `ECC` function (Algorithm 4) is as follows. It takes as input a corresponding state pair $\langle q_{0i}, q_{1j} \rangle$, a path covers P_0 (of M_0) and P_1 (of M_1), a corresponding state pair set W_{csp}, a set of U-equivalent path pairs E_u, a set C-equivalent path pairs E_c, and a *LIST* which maintains a candidate C-equivalent pairs of paths. It returns "*success*" if for every path emanating from q_{0i} an equivalent path originating from q_{1j} is found; otherwise, it returns "*failure*".

The function `checkFalseComputation` returns `True` if the loop at q_{0i} under the propagated condition will execute at least once, over all possible in-

Algorithm 4: ECC($q_{0i}, q_{1j}, P_0, P_1, W_{csp}, E_u, E_c, LIST$)

if q_{0i} is a loop header and doLoopTest[q_{0i}] is TRUE **then**
  doLoopTest[q_{0i}]=FALSE;
  if checkFalseComputation(q_{0i}) returns True **then**
    avoidLoopExitPath[q_{0i}]=TRUE; /* Ignore False Computation */
  end if
end if
foreach path $\beta : (q_{0i} \Rightarrow q_{0m})$ in P_0 **do**
  if q_{0i} is a loop header and avoidLoopExitPath[q_{0i}] is TRUE **then**
    avoidLoopExitPath[q_{0i}]=FALSE;
    continue;
  end if
  if Path β is already present in the $LIST$ **then**
    continue; /* prevent recursions which lead to an infinite loop */
  end if
  $(\beta, \alpha, \vartheta'_{0m}, \vartheta'_{\vartheta_{1n}}) \leftarrow$ findEquivalentPath$(\beta, \vartheta_{0i}, q_{1j}, \vartheta_{1j}, P_0, P_1)$;
  if path $\alpha : (q_{1j} \Rightarrow q_{1n})$ can be found in P_1 such that $\beta \simeq_u \alpha$ **then**
    $E_u = E_u \cup \{(\beta, \alpha)\}$; /* U-equivalence */
    $W_{csp} = W_{csp} \cup \{(q_{0m}, q_{1n})\}$;
  else if path $\alpha : (q_{1j} \Rightarrow q_{1n})$ can be found in P_1 such that $\beta \simeq_c \alpha$ **then**
    if q_{0m} or q_{1n} is reset state **then**
      return *failure*; /* Reset state is reached with mismatch */
    else if q_{0m} or q_{1n} appears as the final state of some path already in $LIST \wedge$ loopInvariant$(\beta, \alpha, \vartheta'_{0m}, \vartheta'_{1n})$ **then**
      return *failure*; /* Propagated values are not loop invariant */
    else
      $\vartheta_{0m} \leftarrow \vartheta'_{0m}$; $\vartheta_{1n} \leftarrow \vartheta'_{1n}$;
      Append $\langle \beta, \alpha \rangle$ to $LIST$
      ECC(q_{0m},q_{1n}, $P_0, P_1, W_{csp}, E_u, E_c, LIST$);
    end if
  else
    return *failure*; /* Equivalent Path of β may not be present in P_1 */
  end if
end foreach
$E_c = E_c \cup \{$Last member of $LIST\}$;
$LIST \leftarrow LIST \backslash \{$Last member of $LIST\}$;
if q_{0i} is a loop header **then**
  doLoopTest[q_{0i}]=TRUE;
end if
return *success*;

Algorithm 5: `loopInvariant`$(\beta, \alpha, \vartheta'_{0m}, \vartheta'_{1n})$

Input : A path $\beta \in P_0$, the propagated vector ϑ'_{0m} at the end state q_{0m} of β, path $\alpha \in P_1$ which is the C-equivalent of β, the propagated vector ϑ'_{1n} at the end state q_{1n} of α

Output: A Boolean value

```
1  foreach variable x in ϑ'_0m and ϑ'_1n do
2      if x is an unmarked variable (Case 1) then
           /* Fig. 3.8 and 3.9 depict the case 1 (scenario S_3)      */
3          if Case 1.1 then
4              Set each unmarked variable to its symbolic value "x";
5          end if
6          if Case 1.2 then
7              one of the tests in Case 1.2 (a) and (b) fails then return False;
8          end if
9      end if
10     if x is a marked variable (Case 2) then
           /* Fig. 3.12 and 3.13 depict this case (scenario S_1, S_2)  */
11         if before entery to the loop x has its symbolic value in one of the loops
             and has the value e_x in other loop then
12             if Case 2.1 or Case 2.2 then
13                 one of the tests in Case 2.1 (a) and (b) fails then return False;
14             end if
15         end if
16         if Case 2.3 then
17             one of the tests in Case 2.3 (a) and (b) fails then return False;
18         end if
19     end if
20 end foreach
21 return True;
```

puts in M_0. It returns `False` otherwise. The function `checkFalseComputation` should be invoked once for all paths that terminate in the state q_{0i}. Moreover, a call to `checkFalseComputation` should be avoided if the state q_{0i} is reached through some back edge. To guarantee this, each loop header is associated with a flag *doLoopTest*. At each loop header state q_{0i}, we also associated a flag *avoidLoopExitPath*. This flag is used to ensure that after avoiding the loop exit path once the loop exit path must be checked for subsequent calls of the function `ECC` for the state q_{0i}.

The function `ECC` invokes the function `findEquivalentPath` to find a U-

or C-equivalent path $\alpha : (q_{1j} \Rightarrow q_{1n})$ in the transformed FSMD M_1 for each path $\beta : (q_{0i} \Rightarrow q_{0m})$ starting from state q_{0i} of the original FSMD M_0. The `findEquivalentPath` function of the EVP method is given in Algorithm 3. The function `findEquivalentPath` returns a 4-tuple $\langle \beta, \alpha, \vartheta'_{0m}, \vartheta'_{1n} \rangle$ where β and α are corresponding paths as described above, ϑ'_{0m} is the propagated vector at the end state q_{0m} of β and ϑ'_{1n} is the propagated vector at the end state q_{1n} of α. If $\vartheta'_{0m} \equiv \vartheta'_{1n}$ then the path α is U-equivalent to path β. Consequently, the data structure W_{scp} gets updated (line 18). If `findEquivalentPath` does not find any path α in M_1 whose condition of execution R_α satisfies either $R_\beta^{\vartheta_{\beta_s}} \equiv R_\alpha^{\vartheta_{\alpha_s}}$, or $R_\beta^{\vartheta_{\beta_s}} \implies R_\alpha^{\vartheta_{\alpha_s}}$ or $R_\alpha^{\vartheta_{\alpha_s}} \implies R_\beta^{\vartheta_{\beta_s}}$, then it returns $\alpha =$ `NULL` (i.e., M_0 and M_1 may not be equivalent, handled in line 30). If $\alpha \neq$ `NULL` and $\vartheta'_{0m} \not\equiv \vartheta'_{1n}$, then the path α is candidate C-equivalent to the path β and hence further value propagation is required. However, the following checks are carried out first and `ECC` reports "*failure*" in the following scenarios:

1. if one of the state q_{0m} and q_{1n} is a reset state (line 21) it returns "*failure*";

2. if a loop has been crossed over then the function `ECC` invokes the function `loopInvariant`. The `loopInvariant` function of the EVP method is given in Algorithm 5. The function `loopInvariant` checks for the loop invariance of the propagated vector ϑ'_{0m} and ϑ'_{1m}. The function `loopInvariant` returns `True` if each marked and unmarked variables satisfy their respective cases as mentioned in Section 3.5.3. If it returns `False` then the function `ECC` returns "*failure*". Note that the function `loopInvariant` trivially returns `True` if case 1.1 in Section 3.5.3 holds for each variable, i.e., the variable is defied identically in both the loops.

If $\vartheta'_{0m} \not\equiv \vartheta'_{1n}$ and the above two cases do not occur, then $\langle \beta, \alpha \rangle$ is appended to *LIST* and the propagated vector at q_{0m} and q_{1n}, are updated and `ECC` calls itself recursively (line 27). It may be noted that while updating the propagated vector (line 25), we update only mismatched variable values and reset the other variable to their symbolic values if the state is the loop header; otherwise, we update all the variable values. When `ECC` reaches line 38, it implies that for every chain of paths emanating from the state q_{0i}, there exists a corresponding chain of paths emanating from q_{1j} such that their final paths are U-equivalent.

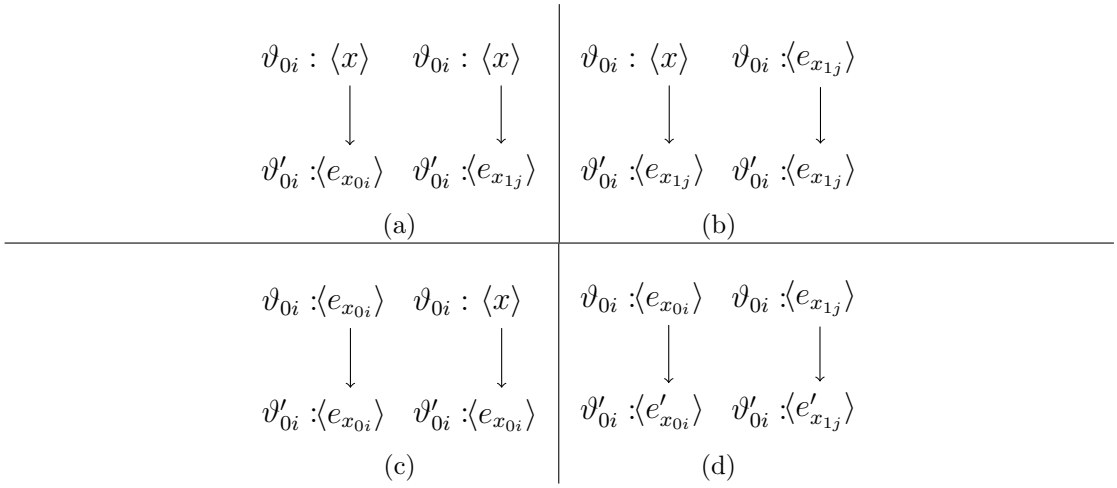


Figure 3.16: All possible scenarios where x has some mismatch at the end of the loop

3.7 Correctness and Complexity

Lemma 1. *If there is a mismatch in the propagated vectors after executing the loop once and Algorithm 5 terminates successfully (at step 21) then x is invariant in the loop.*

Proof. Let q_{0i} be the entry/exit state of a loop body in M_0 and its corresponding state q_{1j} be the entry/exit state of a loop body in M_1. The state q_{0i} has the propagated vector ϑ_{0i} before entering the loop and the propagated vector ϑ'_{0i} after traversal of the path leading to q_{0i}. Similarly, state q_{1j} has the propagated vector ϑ_{1j} before entering the loop and the propagated vector ϑ'_{1j} after traversal of the path leading to q_{1j}. The lemma is proved by case analysis as follow. Let us consider all the possibilities for the variable x as shown in Fig. 3.16.

1. The variable x has its symbolic value in ϑ_{0i} and ϑ_{1j} (i.e., the variable x has the same value in both the loops). After exiting the loop, there is a mismatch for x in ϑ'_{0i} (say $e_{x_{0i}}$) and $\vartheta_{1j'}$ (say $e_{x_{1j}}$) as shown in Fig. 3.16(a). If $e_{x_{0i}}$ or $e_{x_{1j}}$ is not invariant in its corresponding loop then Algorithm 5 returns `False` in step 7.

2. The variable x has its symbolic value at ϑ_{0i} and $e_{x_{1j}}$ at ϑ_{1j}. After executing the loop, the values of x match in both the loops (i.e., x has the same value

($e_{x_{1j}}$) as shown in Fig. 3.16(b). If the expression $e_{x_{1j}}$ is not an invariant in both the loops then Algorithm 5 returns `False` in step 13.

3. The variable x has the value $e_{x_{0i}}$ at ϑ_{0i} and its symbolic value at ϑ_{1j} and after executing the loop the value of x matches in both the loops (i.e., x has the same value ($e_{x_{0j}}$) as shown in Fig. 3.16(c). If the expression $e_{x_{1j}}$ is not an invariant in both the loops then Algorithm 5 returns `False` in step 13.

4. The variable x has the value $e_{x_{0j}}$ at ϑ_{0i} and $e_{x_{1j}}$ at ϑ_{1j}. After exiting the loop the value of x is not the same as before entering the loop as shown in Fig. 3.16(d). Then Algorithm 5 returns `False` in step 17

From the above it is clear that if there is a mismatch in the propagated vectors after executing the loop once and Algorithm 5 terminates successfully (at step 21) then x is invariant in the loop. ■

Lemma 2. *If Algorithm 1 terminates successfully (at step 7) and some code segment in the original behavior M_0 is moved before (after) the loop from inside the loop body in the transformed behavior M_1 then the loop exit path must be a false computation under all propagated conditions at the loop header.*

Proof. Let q_{0i} be the entry/exit state of a loop body in M_0 and its corresponding state q_{1j} be the entry/exit state of a loop body in M_1. Let the definition of the variable x be an invariant in the loop at q_{0i}. It is moved from inside the loop body at q_{1j} to all the paths leading to q_{1j}. Let β be one such path in M_0 which is C-equivalent to the path α ($\beta \simeq_C \alpha$) in M_1 (i.e., there is a mismatch for the variable x). Before entering the loop, let the propagated condition at q_{0i} be CP_β through the path β in M_0. If the function `checkFalseComputation` returns `Flase` under the CP_β then the loop exit path at q_{0i} is not a false computation. In this case the function `enhancedCorrespondenceChecker` does not ignore the loop exit path and finds the C-equivalent path in the transformed M_1 (i.e., there is a mismatch for the variable x). Since the mismatch for x persists till the reset state, *enhancedCorrespondenceChecker* calls itself recursively until the reset state is reached and returns *failure* (i.e., M_0 is not equivalent to M_1) in step 21.

Let the definition of x be moved from inside the loop body at q_{1j} to the loop exit path emanating from q_{1j}. Let the loop exit path at q_{0i} be not a false computation under some propagated condition at q_{0i}. In this case also the function

`enhancedCorrespondenceChecker` does not ignore the loop exit path and finds the C-equivalent path in the transformed M_1 (i.e., there is a mismatch for the variable x) and calls itself recursively until the reset state is reached and returns *failure* (i.e., M_0 is not equivalent to M_1) in step 21. ■

3.7.1 Soundness

The paper [42] shows the correctness of the VP method assuming that the pair of paths declared as U-equivalent or C-equivalent by the function `findEquivallentPath`, are indeed U-equivalent or C-equivalent. But this may not always be true as shown in Example 3. Example 3 shows a case where the VP method provides a false positive result. Section 3.5 proposed a solution to show the non-equivalence in the case given in Example 3.

Theorem 2 (Partial correctness)**.** *If the verification method terminates at step 7 of Algorithm 1, then* $M_0 \sqsubseteq M_1$.

Proof. If the verification method terminates at step 7 of the function `containmentChecker` then we need to prove the following claims.

1. The set $E = E_u \cup E_c$ contains a member for each path in the path cover P_0.
2. All paths of the path cover M_0 leading the reset state of M_0 will have a U-equivalent path in the path cover in P_1 leading to the reset state of M_1.
3. If there is a loop invariant code motion from inside the loop then it is a valid code motion.

The set E_u contains the pair of U-equivalent paths, and the set E_c contains the pair of C-equivalent paths. A pair of paths is added to the set E_u and E_c at step 17 and 33, respectively, of `enhancedCorrespondenceChecker`. Now the pair of paths declared by `findEquivallentPath` is actually U-equivalent and C-equivalent and added to the set E_u and E_c respectively. The fact that $E = E_u \cup E_c$ contains a member for each path in the path cover P_0 can be proved in a way similar to the method in [42].

Claim 2 can be proved by contradiction. Let the paths $\beta \in P_0$ and $\alpha \in P_1$ lead to the reset states of M_0 and M_1, respectively and $\beta \simeq_c \alpha$. In this case, the function `enhancedCorrespondenceChecker` returns *failure* to `containmentChecker`

as shown at step 21; consequently, `containmentChecker` terminates at step 3, not at step 7, is a contradiction.

If there is a loop invariant code motion from inside the loop then Lemma 1 and 2 ensure the validity of code motion. ■

3.7.2 Termination

Theorem 3 (Termination)**.** *Algorithm 1 always terminates.*

Proof. The function `loopInvariant` terminates since it involves just a comparison of two propagated vectors. The function `findEquivalentPath`$(\beta, q_{1j}, \cdots)$ tries to find a path α starting from $q_{1j} \in M_1$ such that $\beta \simeq_u \alpha$ or $\beta \simeq_c \alpha$. It checks all the transitions from q_{1j} in the worst case. Hence it terminates as well. In Algorithm 8, the outermost loop (7–32) of the function `enhancedCorrespondenceChecker` is executed only $|P_0|$ (number of elements in P_0) time which is finite. In Algorithm 8 `enhancedCorrespondenceChecker` can invoke itself recursively. The `enhanced−CorrespondenceChecker`(q_{0i}, q_{1j}) invokes itself with the end state of some path β emanating from q_{0i} and some path α emanating from q_{1j}. If the end state of path β or path α is a reset state then `enhancedCorrespondenceChecker`(q_{0i}, q_{1j}) returns *failure* (at step 21). Since the recursive call of the function `enhancedCorrespondence−Checker` does not extend beyond the reset state and the function `enhanced−CorrespondenceChecker` avoids traversing the loop twice (at step 12) the function invokes itself recursively only a finite number of times. ■

3.7.3 Complexity

The condition of execution and data transformations of a path is represented in normalized sum form [101]. The complexity of normalization of a formula F is $O(2^{|F|})$, where $|F|$ denotes the length of the formula. It may be noted that if the number of states in the original FSMD M_0 is n, then the number of states in the transformed FSMD is in $O(n)$. Let n be the number of states in the FSMD and K be the maximum number of parallel edges between any two states. The complexity of the overall verification method is of order of the product of the following two terms:

1. The first term is the complexity of `findEquivalentPath`$(\beta, q_{1j}, \cdots)$. In worst case, the function checks all transitions from q_{1j} to find a path α starting from $q_{1j} \in M_1$ such that $\beta \simeq_u \alpha$ or $\beta \simeq_c \alpha$. The complexity of finding the path α is $O(2^{|F|} \cdot k \cdot n)$. On finding a C-equivalent path, value propagation is carried out in $O(2^{|F|} \cdot |V_0 \cup V_1|)$ time. Hence the overall complexity of `findEquivalentPath`$(\beta, q_{1j}, \cdots)$ is $O(2^{|F|} \cdot (k \cdot n + |V_0 \cup V_1|))$.

2. The second term is of order of the product of the following two terms:

   (a) The number of times *enhancedCorrespondenceChecker* is called from `containmentChecker`. For every element of $Wscp$, the corresponding state set, `containmentChecker` calls `enhancedCorrespondenceChecker`. The maximum number of elements in W_{csp} is $O(n)$.

   (b) The number of times `enhancedCorrespondenceChecker` calls itself recursively. In the worst case, all the states of M_0 can be cut-points. In this case `enhancedCorrespondenceChecker` can recursively call itself $k \cdot (n-1) + k^2 \cdot (n-1) \cdot (n-2) + \cdots + k^{n-1} \cdot (n-1) \cdot (n-2) \ldots 2 \cdot 1 \simeq k^{n-1} \cdot (n-1)^{n-1}$ times.

   Therefore the complexity of the overall method is $O(2^{|F|} \cdot (k \cdot n + |V_0 \cup V_1|) \cdot n \cdot k^{n-1} \cdot (n-1)^{n-1})$. If we ignore the time taken by the SMT solver Z3 then the worst case complexity of the presented method is the same as that of [42].

3.8 Experimental Results

Our equivalence checking algorithm has been implemented in C, and all experiments have been conducted on a laptop with Intel Core 2 Duo processor with 2 GHz and 3GB of RAM. In our experimental setup, loop information i.e., nesting depth and loop header are extracted during construction of the FSMD from the input behavior using dominator tree analysis [102]. All the benchmarks listed in Table 3.1 are taken from [42]. The benchmarks selected such as TLC and GCD are control-intensive design; DCT, DIFFEQ and EWF are data-intensive and LRU is both data and control intensive. The transformed FSMD is obtained from the original one in two steps. First we obtained the intermediate transformed FSMD

Table 3.1: Experimental results on the benchmarks presented in [42]

Benchmarks	M_0		M_1		#Loop	VP		EVP	
	#State	#Path	#State	#Path		Equi	T (ms)	Equi	T (ms)
TLC	13	20	7	16	0	Yes	52	Yes	52
DCT	16	1	8	1	0	Yes	116	Yes	120
EWF	34	1	36	1	0	Yes	40	Yes	44
PERFECT	6	7	4	6	1	Yes	24	Yes	40
GCD	8	11	14	8	1	Yes	56	Yes	116
MODN	8	9	9	9	1	Yes	92	Yes	176
DIFFEQ	15	3	9	3	1	Yes	28	Yes	32
LRU	33	39	32	39	8	Yes	364	Yes	1204
IEEE-754	55	59	44	50	7	Yes	482	Yes	2080
BARCODE	32	55	24	57	15	Yes	540	Yes	4130

Table 3.2: Experimental results on the benchmarks presented in [42]

Benchmarks	VP		EVP	
	Equivalent	Time (ms)	Equivalent	Time (ms)
TLC	No	60	No	64
DCT	No	124	No	128
EWF	No	40	No	44
PERFECT	No	28	No	40
GCD	No	44	No	36
MODN	No	92	No	72
DIFFEQ	No	16	No	16
LRU	No	202	No	840

by running the SPARK tool on these benchmarks. We forced SPARK to apply the code transformation like copy and constant propagation, common sub-expression elimination, and dead code elimination (DCE) to the original behavior to produce the corresponding optimized transformed behavior. The intermediate transformed FSMD obtained by SPARK is converted into the final transformed FSMD according to path-based scheduler. This help us to confirm that our method shows equivalence correctly when control structure has been modified as well as code

motions have arisen.

In our first experiment, we compare our EVP method with the VP method to verify the benchmarks listed in Table 3.1. The objective is to show that our EVP method can prove the equivalence for these benchmarks. Also, we want to compare the execution time (in milliseconds (ms)) with the VP method. The results of these experiments are tabulated in Table 3.1. Our method is able to establish the equivalence in all the benchmarks. Note that in Table 3.1, if a benchmark has no loop then the execution time obtained by our EVP method is almost the same as the VP method. However, when a benchmark has some loop then our method needs more time since at each loop header we invoke the SMT solver Z3 to check whether the loop will execute at least once. For example, LRU benchmark has 8 loops. Therefore our method takes more time as compared to the VP method. This extra check is required to overcome some of the limitations of the VP method as discussed in Section 3.4. Our next experiment will justify this. Table 3.1 confirms that our method is capable of handling all code transformation techniques mentioned here. Since our objective is to compare the run time of our method with the VP method in all the scenarios which the VP method can handle, we prevent SPARK from applying loop invariant code motion transformation to obtain the optimized transformed behavior. It has been shown in experiment 3 that the VP method cannot handle LICM transformation.

In our second experiment, we manually introduce some changes in the benchmarks listed in Table 3.1 so that their original and transformed FSMDs become inequivalent. These modified benchmarks are listed in Table 3.2. The objective of this experiment is to show that our method does not produce false positive results in non-equivalence cases. As shown in Table 3.2 both the methods reported non-equivalence in all these scenarios. Also, the run time of both the method is almost the same except LRU. The experiments 1 and 2 confirms that both the methods are able to show the equivalence correctly.

In our third experiment, we take some of the test-suite distributed with LLVM [103]. These benchmarks contain some loop invariant operations. We forced SPARK to apply LICM transformation to obtain the transformed behavior so that it can hoist loop invariant code before the loop in the transformed behavior. These test cases represent the scenarios S_2 and S_3. The results of these experiments are tabulated in row 1–4 of Table 3.3. From Table 3.3, it is evident that

Table 3.3: Experimental results on test cases where the VP method fails

Benchmarks	VP		EVP	
	Equivalent	Time (ms)	Equivalent	Time (ms)
simple_types_ loop_invariant	No	4	Yes	12
mandel	No	4	Yes	16
mandel2	No	4	Yes	16
himenobmtxpa	No	4	Yes	20
Test 1	Yes	8	No	8
Test 2	Yes	8	No	8
Test 3	Yes	12	No	12
Test 4	Yes	16	No	16

our proposed method can correctly identify the equivalence even when some loop invariant operation `op` is moved before (after) the loop from inside it. However, the VP method reports may not equivalent in these cases. The VP method takes only 4ms for the benchmarks listed in rows 1–4 Table 3.3 because at the loop header it select loop exit path first and shows the non-equivalence immediately. The experimental evaluation shows that our method outperforms the VP method in handling the LICM transformation.

In our fourth experiment, we have created some test cases where the VP method provides a *false positive* results, but our EVP method can prove the non-equivalence. Since we are not able to generate our desired transformed behavior using SPARK, the benchmarks tabulated in row 5–8 of Table 3.3 are manually scheduled. The result of this experiment confirms that the VP method incorrectly reports equivalence for these test cases while our EVP method correctly proves the non-equivalence for these test cases.

In our fifth experiment, some larger benchmarks from CHStone [55] and Bambu HLS tool [14] are selected to evaluate the scalability of our EVP method. The synthesizer used in our experiments is Bambu. The FSMDs are extracted from the behaviors at the input and the output of the scheduling phase of Bambu. We use the function `BF_cfb64_encrypt` in BLOWFISH, the function `Gsm_LPC_Analysis` in GSM, and the function `encrypt` in AES as a source behavior. The results of this

Table 3.4: Experimental results on the benchmarks presented in CHStone benchmarks [55] and the benchmarks listed in Bambu HLS tool [14]

Benchmarks		#c	EVP			
			#Path		Equivalent	Time(ms)
			M0	M1		
Bambu	WAKA	35	4	3	Eq	75
	ARF	43	5	5	Eq	400
	MOTION	44	1	1	Eq	70
CHStone	BLOWFISH	151	21	21	Eq	1519
	GSM	240	96	86	Eq	7152
	MIPS	259	77	51	MNEq	123
	AES	330	132	96	MNEq	857

MNEq: M_0 and M_1 "May Not be Equivalent".
#c: # of lines in c program.
T: Time in milliseconds(ms).

experiment are tabulated in Table 3.4. It is evident from this experiment that the EVP approach can handle the larger benchmarks but fails to show the equivalence for the benchmarks AES and MIPS. We observe that Bambu modifies the control structure significantly. For the benchmarks AES and MIPS, the transformed behaviors represent the scenario where a path in original behavior has been split into more than one path to improve the conditional hardware reuse. The EVP method fails to handle the path split/merge scenario. In the next chapter, we have presented a PBEC approach that can handle the path split/merge scenario.

During our experimentation, we found a bug in the SPARK tool. The bug is in the implementation of the LICM algorithm in the SPARK scheduler. A simple instance of the bug is reported in Fig. 3.17. It may be noted that the transformed behavior is obtained by applying only LICM technique with SPARK. Here the operation $x = 5$ is moved before the loop body in the transformed behavior since the operation $x = 5$ is invariant within the loop in the input behavior. However, the output will not be the same for any input $n \leq 4$. For example when $n = 3$, the value of *out* is zero in the input behavior and its value is five in the transformed

```
int main(){
 int x,i,n,z=0,out;
 x=0;
 for(i=4;i<n;i++){
  x = 5;
  z=z+x;}
 out=z+x;
 return out;}
```

(a) Input Behavior

```
int main(void){
 int x,i,n,z,out,sT0_5;
 int returnVar_main;
 z = 0;x = 0;
 i = 4;x = 5;
 do{
  sT0_5 = (i < n);
  if (sT0_5){
   z = (z + x);
   i = (i + 1);}
  else break;
 }while (1);
 out = (z + x);
 returnVar_main = out;
 return returnVar_main;}
```

(b) Transformed Behavior

Figure 3.17: A bug in SPARK

behavior. This behavior is proved to be non-equivalent by our EVP method. Our method finds a previously unknown bug in a widely used HLS framework indicates the usefulness of our method.

3.9 Conclusion

In this chatper, we have presented an enhanced VP method for code motion involving loops. Like the VP method, our method is also capable of handling control structure modification of input behavior and uniform and non-uniform code motion and code motion across loops. In addition, our method can also handle the scenario where some code segment is moved before (after) the loop from inside the loop body. In addition, our method is capable of automatically identifying false computations and prove non-equivalence of FSMDs for the cases where the VP method provides *false positive* results. Our method discovered a bug in SPARK that long-term use did not uncover.

Chapter 4

Verification of Scheduling of Conditional Behaviors in High-level Synthesis

4.1 Introduction

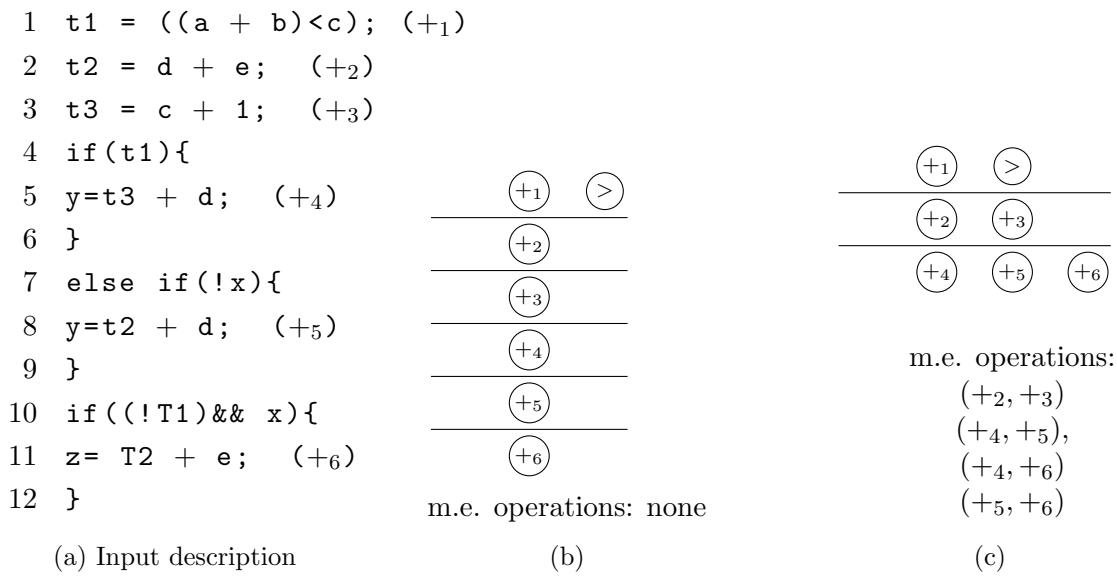
4.1.1 Scheduling of Conditional Behaviors

In general, the major tasks of HLS includes scheduling operations from the given behavioral description into control steps under the required timing and hardware resources constraints. High-level synthesis schedulers can take advantage mutual exclusivity of operations and schedule two mutually exclusive operations in the same cycle on the same resource. Two operations in a behavioral description are mutually exclusive if the results of the two operations are never needed together in the execution of the system. Mutually exclusive operations can be implemented with the same hardware component and scheduled in the same cycle. Consequently, total delay of the schedule and the resource usage is reduced. Consider the following Fig. 4.1(a) description written in C language. Suppose that we want to implement this system using only one adder and one comparator and suppose that these functional units (FUs) take one control step to execute the operation. Assuming we can not identify any mutually exclusive operators, then six cycles are needed at least to complete the behavioral description as show in Fig 4.1(b). However if we can identify that the pairs $(+_2, +_3)$, $(+_4, +_5)$, $(+_4, +_6)$, and $(+_5, +_6)$ are all possible mutually exclusive operations in the example Fig. 4.1(a). Now three cycle are need with the same number of FUs to complete the algorithm as shown in Fig. 4.1(c). The information about the mutually exclusive pairs of operation is very useful to decrease an amount of hardware required to implement specification

without increasing the latency that is, the conditional reuse of resources. In literature several approaches have been reported to identify the largest set of mutually exclusive operation pairs. However, the possibility of conditional reuse depends not only on the number of mutex operations pairs detected by an algorithm, but also on the way in which specifications are written by designers. The optimization techniques such as in [44] transforms the input description to improve the possibility of conditional reuse of resources depends on the number of mutually exclusive pairs of operations. Therefore, it is necessary to verify the semantic equivalence between the original and the transformed behaviors.

4.1.2 Summary of Verification of Scheduling of Conditional Behaviors

Path-based equivalence checking approaches [37–42,56] have made significant progress in the verification of the scheduling phase of HLS. These translation validation approaches are useful since they can verify that the correct code resulted from various compiler optimization techniques applied in each iteration of the scheduling phase of HLS without unrolling the loops. However, they (including this work) cannot verify the correctness of the scheduling phase. A PBEC approach based on value



(a) Input description (b) (c)

Figure 4.1: An example of behavioral description

propagation [42], for example, can verify the code motion involving loops. The behaviors are modeled as an FSMD in PBEC approach. These approaches break down an FSMD into smaller segments by introducing cutpoints so that each loop in the FSMD is cut by at least one cutpoint. This is based on the Floyd-Hoare method of program verification [54]. The set of all paths from a cutpoint to another cutpoint without any intermediate occurrence of a cutpoint is a path cover of the FSMD. PBEC approaches establish the equivalence between two behaviors by showing the equivalence between the paths present in the path cover of these two behaviors. The primary focus of the existing PBEC approaches is on ensuring that the data dependencies are not violated due to scheduling of operations and transformation of behaviors due to application of various compiler optimizations [15] applied during the scheduling phase of HLS. Equivalence of two programs over Integers is inherently undecidable [45]. Therefore, existing PBEC approaches may produce false negative results.

4.1.3 Contributions

In this work, we identify some limitations of the existing PBEC approaches. Specifically, we identify the optimization techniques such as in [44] which split a paths into multiple paths in the scheduled behavior. In this case, existing PBEC approaches [37–42, 56] fail to show the equivalence. In addition, PBEC approaches based on value propagation also fail to show the equivalence for some scenarios where conditional blocks having an equivalent conditional expression are combined into one conditional block. Moreover, we identify that the cutpoint selection scheme in PBEC approaches are too restricted to handle control structure related transformations.

In this work, we present a PBEC approach based on value propagation to overcome these limitations without affecting the power of existing approaches. Specifically, the contributions of this work are as follows:

1. We redefine the notion of the equivalence of paths in PBEC approach to handle path split/merge.

2. We also present a new cutpoint selection scheme to handle control structure related transformations in PBEC without much performance overhead.

3. We present a PBEC method to verify the scheduling of conditional behaviors without affecting the power of the state-of-the-art existing approach [56]. This method is also capable of handling merging of conditional blocks.

4. We implement our proposed method and demonstrate its usefulness with experimental results.

The rest of this chapter is organized as follows. Motivating examples highlighting the limitations of the state-of-the-art PBEC approaches are given in Section 4.2. A solution to overcome the current limitations of PBEC approaches are presented in Section 4.3. The notion of equivalence of two paths is introduced in Section 4.4. The overall verification process is presented in Section 4.5. The correctness and complexity of the proposed method are discussed in Section 4.6. Experimental results are given in Section 4.7. Section 4.8 concludes the chapter.

4.2 Motivations

In this Section, we represent the scenarios where the state-of-the-art PBEC approaches fail to show the equivalence even though behaviors are equivalent. We then propose the solutions in the next section.

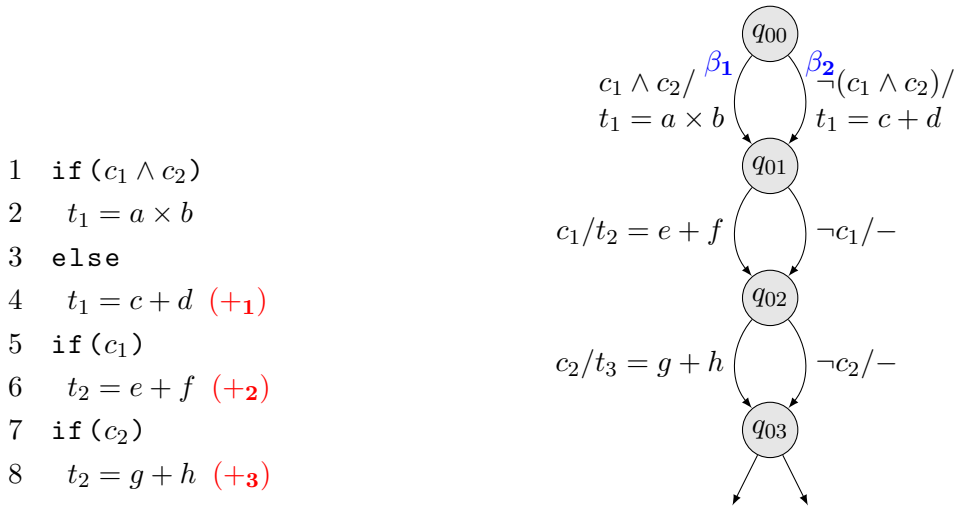
4.2.1 Path Split

To improve the conditional hardware reuse in HLS, the approach presented in [44] transforms the original behavior in Fig. 4.2(a) (and its corresponding FSMD in Fig. 4.2(b)) into the equivalent one in Fig. 4.2(c), where the condition $(c1 \wedge c2)$ has been split[1]. As a result, the path $\beta_1 = \langle q_{00} \xrightarrow{c_1 \wedge c_2} q_{01} \rangle$ in M_0 is equivalent to path $\alpha_1\alpha_3$[2] in M_1, and the path β_2 in M_0 is equivalent to the union of the paths[3] $\alpha_1\alpha_4$ and α_2 in M_1 i.e, $R_{\beta_2} \equiv R_{\alpha_1\alpha_4} \vee R_{\alpha_2}$ and $s_{\beta_2} = s_{\alpha_1\alpha_4} = s_{\alpha_2}$. Let us consider that we have only one adder and one multiplier, and these function units take one control step to execute the corresponding operation. In this case, there is at least

[1]This example is taken from [44]

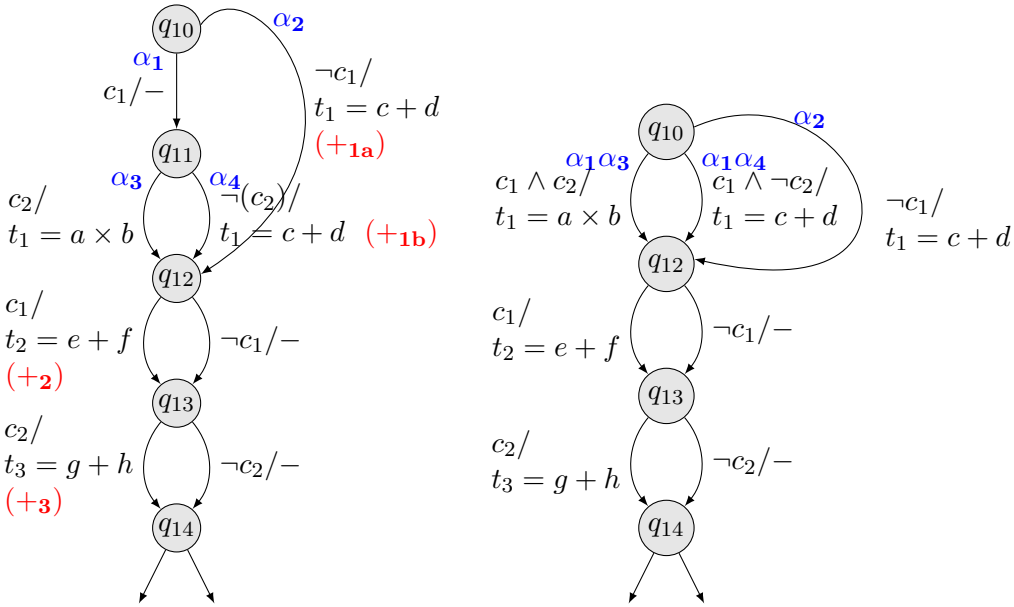
[2]$\alpha_1 : \langle q_{10} \xRightarrow{c_1} q_{11} \rangle$, $\alpha_3 : \langle q_{11} \xRightarrow{c_2} q_{12} \rangle$, $\alpha_1\alpha_3$ reports the concatenated paths α_1 and α_3.

[3]union of paths, say β_i, β_j, emanating from the state q_i and ending at q_j and have the same data transformation, represents a path where condition of execution is $R_{\beta_i} \vee R_{\beta_j}$ and data transformation is the same as path β_i.



(a) Input description

(b) M_0: Original behavior



(c) M_1: Scheduled behavior obtained by [44]

(d) M_2: Transformed behavior

Figure 4.2: Transformations on the input description to enhance the conditional hardware reuse

three cycles needed to complete all the operations in M_0. It may be noted that the operation pair $(+_1, +_2)$ in Fig. 4.2(a) cannot share the functional unit because if

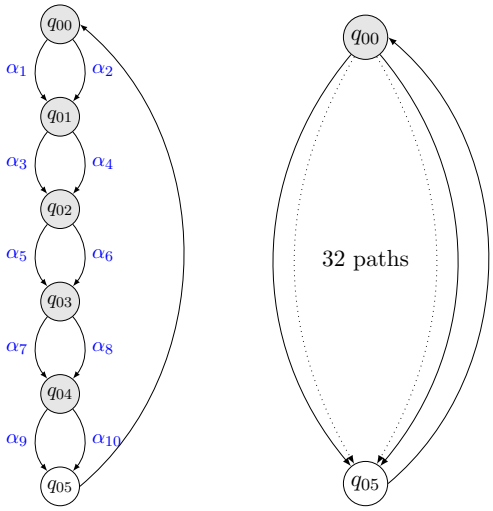


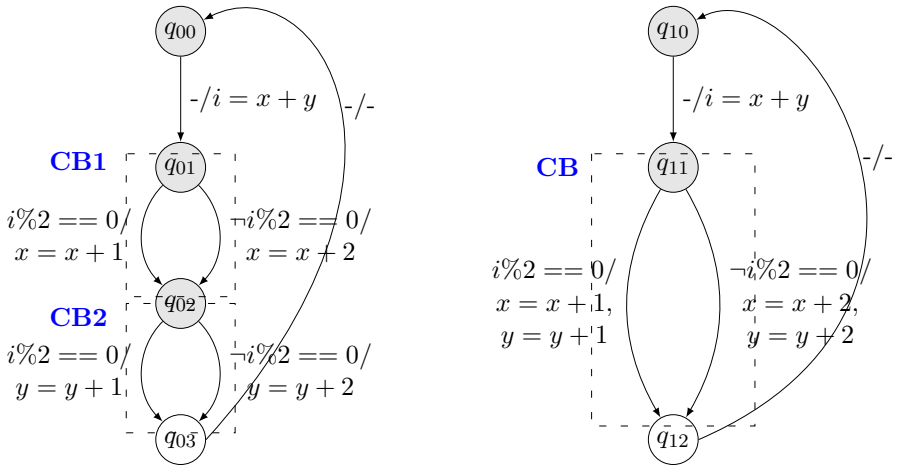
Figure 4.3: A cutpoint example (a) An FSMD M_0 with all the states as cutpoints; (b) An FSMD M_0 with minimal cutpoint

c_1 is true and c_2 is false, both operations are needed. Similarly, an operation pair $(+_1, +_3)$ in Fig. 4.2(a) is not mutually exclusive and cannot share the functional unit because if c_1 is false and c_2 is true both operations are needed. To achieve two cycles, the paths in the original behavior have been split in the transformed behavior shown in Fig. 4.2(c). With this conditional transformation, the operation pair $\{+_{1a}, +_2\}$ can share a functional unit. Similarly, the operation pair $\{+_{1b}, +_3\}$ can also share a functional unit. Therefore, the behavior can be scheduled in two cycles.

During equivalence checking the PBEC approaches fail to handle the scenario where a path in an FSMD is equivalent to the union of the paths in another FSMD. Therefore, all existing PBEC approaches fail to find the equivalent paths in the FSMD M_1 for the path β_2 in FSMD M_0. As a result, they report that behaviors "May Not be Equivalent". In Subsection 4.3.1, we propose a solution to handle this type of scenario.

4.2.2 Choice of Cutpoints

A PBEC approach obtains the path cover by introducing the cutpoints. The minimum number of cutpoints are those who cut each loop in an FSMD by at



(a) M_0 (b) M_1

Figure 4.4: An example of `if` optimization

least one cutpoint. Other cutpoints are redundant one, introduced to improve the runtime of the PBEC approach. Most of the PBEC approaches [37–42, 56] find a path cover by setting the reset state and the branching states, i.e., states with more than one outgoing transition, of the FSMD as cutpoint. These approaches select all the branching states as cutpoint for the FSMD in Fig. 4.3(a) therefore there will be 10 paths (i.e., $\alpha_1, \alpha_2, \cdots, \alpha_{10}$) in the path cover. Instead, if we choose minimal cutpoint, then only the reset state q_{00} will be the cutpoint and there would be 32 paths in the path cover as shown in Fig. 4.3(b). The selection of additional cutpoints helps these approaches to reduce the size of the path cover exponentially and improve their runtime.

However, choosing all branching states as cutpoint poses problem in finding the path-based equivalence where the transformed behavior is obtained by splitting the paths in original behavior as shown in Fig. 4.2. The PBEC approaches select the states q_{00}, q_{01}, q_{02}, and q_{03} in M_0 in Fig. 4.2(b) and q_{10}, q_{11}, q_{12}, q_{13}, and q_{14} in M_1 shown in Fig. 4.2(c) as cutpoints. The selection of the state q_{11} in M_1 as a cutpoint makes the `if-else` block unbalanced and makes difficult to show the path level equivalence for a PBEC approach. If we do not select q_{11} as cutpoint then resulting transformed behavior M_2 is shown in Fig. 4.2(d). Now from the state q_{10}, there will be three paths $\alpha_1\alpha_3$, $\alpha_1\alpha_4$ and α_2. The path β_1 in M_0 will be equivalent

to the path $\alpha_1\alpha_3$ in M_2. The path β_2 in M_0 will be equivalent to the union of α_2 and $\alpha_1\alpha_4$. Thus, avoiding q_{11} as cutpoint simplifies the control structure of M_1 and would help the PBEC approach to show the equivalence. However, as discussed previously, the PBEC approach cannot handle the scenario where a path in an FSMD is equivalent to the set of the paths in another FSMD. Therefore, they fail to show the equivalence between M_0 and M_2 as well. In the Subsection 4.3.2, we propose a new cutpoint selection scheme to simplify the `if-else` block. We then use the solution presented in Subsection 4.3.1 to handle path split.

4.2.3 `If` Optimization

Conditional blocks, not necessary to be adjacent, (generally represented as `if-else`) having an equivalent conditional expression can be combined into one conditional block. This reduces the number of condition checking and the total number of states in transformed behavior. Thus, it reduces the number of paths in the path cover. For example, consider the behaviors in Fig. 4.4. Here two conditional blocks (CB) denoted as CB1 and CB2, in M_0 in Fig. 4.4(a) are merged into one conditional block CB in M_1 shown in Fig. 4.4(b).

The path extension based approaches in [38, 39] fail to verify code motion across loops. Therefore, these methods cannot not handle the scenarios where conditional blocks are merged across the loop. The VP [42] and the EVP [56] can handle code motion across the loops. Therefore, they can show equivalence when conditional blocks are merged across the loops. However, we identify that they fail when conditional merge leads to the reset state as shown in Fig. 4.4. It may be noted that the VP and the EVP method fail whenever they reach the reset state of one FSMD and do not reach the reset state of the other FSMD during the course of equivalence checking. These approach find that the path $q_{01} \xrightarrow{i\%2==0} q_{02}$ is not equivalent to the path $q_{11} \xrightarrow{i\%2==0} q_{10}$ and needs to propagate the mismatched value in the subsequent paths in q_{02} and q_{10}. However, the state q_{10} is the reset state. Hence, the method reports behaviors "May Not be Equivalent". In Subsection 4.3.3 we present a method to handle this type of scenarios involving `if` optimization.

Table 4.1: Comparing the effect of cutpoint selection criteria on the performance of the PBEC approach presented in [56]

Benchmarks	#C-line	#State	#Cutpoint			#Paths			Time(ms)		
			S_1	S_2	S_3	S_1	S_2	S_3	S_1	S_2	S_3
EX1	33	21	7	1	3	13	34	20	16	57	22
EX2	41	30	13	1	4	25	125	16	13	160	19
EX3	80	51	19	1	3	37	192	29	18	215	26
EX4	73	56	24	1	3	47	260	58	24	355	51
EX5	60	33	12	1	7	23	486	18	17	711	15
EX6	358	240	86	19	54	171	584	139	73	1938	68
EX7	204	134	49	5	30	97	981	84	33	6745	43

4.3 Proposed Solution

4.3.1 Handling Path Split

Consider the input behavior M_0 and its transformed behavior M_2 in Fig. 4.2. It may be noted that the path α_2 and the path $\alpha_1\alpha_4$ consist of the same operation list and the disjunction of the conditions of execution $R_{\alpha_2} = c_1 \wedge \neg c_2$ and the condition of execution $R_{\alpha_1\alpha_4} = \neg c_1$ is equivalent to the condition of execution $R_{\beta_2} = \neg(c_1 \wedge c_2)$ of the path β_2, i.e., $(R_{\alpha_2} \vee R_{\alpha_1\alpha_4}) \equiv R_{\beta_2}$. In the search for the corresponding equivalent path of β_2, if we compare it with α_2 then $R_{\beta_2} \not\equiv R_{\alpha_2}$, but it may be observed that $R_{\alpha_2} \implies R_{\beta_2}$. This shows that R_{α_2} is a stronger condition than R_{β_2} and also indicates that β_2 has been split into more than one path in M_2 where α_2 is one path among these paths. We find the remaining paths in M_2 with the updated condition $\neg c_1$ (where $\neg c_1 \equiv R_{\beta_2} \wedge \neg R_{\alpha_2}$) for β_2. We find that $\alpha_1\alpha_4$ is equivalent to β_2 with the updated condition. Hence, the β_2 has been split into two paths α_2 and $\alpha_1\alpha_4$ and it is equivalent to the union of these two paths. In general, for two paths β in M_0 and α in M_1 if $R_\alpha \implies R_\beta$ then first we check whether the path β of M_0 has been split into multiple paths in M_1. The path α is one path and we search the remaining paths in M_1 with the updated condition $R_\beta \wedge \neg R_\alpha$ of the path β. However, if we fail to find the path, then it is not the case of path splitting. This scenario still can be handled with usual value propagation, as discussed in Section 4.4.

4.3.2 Cutpoint Selection Scheme

The choice of cutpoints is not unique and it is not guaranteed that path covers obtained from any choice of cutpoints in both the FSMDs result in equivalence of FSMDs. The same situation arises for the behaviors M_0 and M_1 in Fig. 4.2 when the PBEC approaches select all the branching states as cutpoint. If we remove the end state of the path α_1 as cutpoint then it simplifies an `if-else` block. However, any loop node (i.e., entry point of the loop) must be kept as cutpoint even inside an `if-else` block. Thus, within an `if-else` block, we should not designate all the internal branching nodes as cutpoints except the loop nodes. Another option of simplifying an `if-else` block is by selecting only loop nodes as cutpoints but this increases the number of paths in the path cover exponentially. Therefore, we can find the cutpoints for a given FSMD by using one of the selection criteria:

- S_1: Select the reset state and all branching states as cutpoint.
- S_2: Select the reset state and only loop states as cutpoint.
- S_3: Select the reset state and all branching states as cutpoint except the internal nodes of an `if-else` block which are not a loop node.

The PBEC approaches [37–42,56] use cutpoint selection criteria S_1. The scheme S_2 is the minimum number of cutpoints based on the Floyd-Hoare method of program verification [54]. In this work, we proposed a new cutpoint selection scheme S_3. We design some test cases which contains several unbalanced `if-else` blocks. We modify the EVP method presented in previous chapter so that it works on each cutpoint selection criteria mentioned above. We want to compare the runtime effects of cutpoint selection criteria. Therefore, we run the value propagation based PBEC approach, where the input and transformed behavior are the same for each criterion. The result of this experiment is tabulated in Table 4.1. The second column represents the number of lines in the corresponding C code for each test case. The third column represents the number of states in the corresponding FSMD for each C code. The number of cutpoints, the number of paths in the path cover and the runtime in milliseconds (ms) of the EVP method for each case is listed in the columns $4-6$, $7-9$ and $10-12$, respectively. Note that in rows $1-5$ of Table 4.1, the number of cutpoints for S_1 denotes the total number of `if-else` blocks for a given test case and the number of cutpoints for S_3 denotes the number

of `if-else` blocks which is not defined inside of any `if-else` blocks for a given test case. The same for S_2 gives the number of loops in the test cases. It is evident from the results in Table 4.1 that for the selection criteria S_2 the number of paths in the path cover is on average almost 7 times higher then paths for S_1. Hence, it needs more time to show the equivalence as compared to others. However, the number of paths and hence the runtime for the S_1 and S_3 are comparable. Note that the time reported here is the minimum since there is no value propagation is required. However, this study gives a trend of the runtime for all the selection criteria.

As expected, for the selection criteria S_1 the number of paths in the path cover is least as compared to other criteria. Therefore, it needs less time to show equivalence. If we use the selection criteria S_2, then the number of paths in the path cover is more. Hence, it needs more time to show the equivalence as compare to the other two approaches. If we apply selection criteria S_3 then the number of paths in the path cover also marginally increases as compared to selection criteria S_1. Hence, it needs extra time to show the equivalence as compare to selection criteria S_1. From the runtime shown in Table 4.1, it is clear that the criteria S_3 simplifies the control structure of a given FSMD in the cost of marginal additional time. The EVP method may consider a path multiple times (by the concept of value propagation) when the transformed behavior is obtained by applying the code motion techniques on the input behavior. Thus, the runtime to show the equivalence increase for the EVP method and the time shown in Table. 4.1 for the EVP method is the minimum. However, this study gives a trend of runtime for all the selection criteria.

The cutpoint selection scheme S_2 is not feasible in practice since this increase the size of the path cover exponentially as shown in the Fig. 4.3(b). The internal structure of a nested `if-else` block are merged due to control structure related transformation as shown in Fig. 4.2. The cutpoint selection scheme S_1 selects all branching points as cutpoints. The paths this scheme identifies inside such nested `if-else` block in both the behaviors, cannot be co-related by the equivalence checking method. With S_1, therefore, the EVP cannot show equivalence of such split/merged paths. Our proposed scheme S_3 does not consider internal branching states inside a nested `if-else` block as cutpoints. Therefore, paths identified by S_3 inside such nested `if-else` block in both the behaviors can be co-related.

On top of that, we need the path split equivalence as discussed in previous subsection. Therefore, the cutpoint selection scheme S_3 and the path split equivalence are necessary for the handling of split/merged paths for our approach presented in 4.5. In our experimental section we have also discussed the runtime performance of the EVP method with S_1 and S_3 on benchmarks without path split/merge.

4.3.3 Handling the Scenario Involving `if` Optimization

The value propagation method [42,56] uses the concept of *null path* to handle the conditional merge scenario. A null path (of length 0) from a state q to the same state q has the condition of execution `T` and a null (identity) data transformation. We denote a null path emanating from a state q as η_q. As discussed in Subsection 4.2.3 the value propagation method [42,56] cannot handle the condition merge lead to the reset state as shown in Fig. 4.4. In the course of equivalence checking if these methods reach at the reset state in one of the behavior (say M_0) and do not reach the reset state in other behavior (say M_1) then they should do the following:

1. They should consider the *null* path at the reset state of M_0.
2. They must proceed the equivalence checking until condition at null paths matches with the path in FSMD M_1, or the reset state of M_1 have reached.

This strategy helps to find the equivalence between the behaviors shown in Fig. 4.4. The function `findEquivalentPathAtReset` is used to handle this scenario in our proposed method.

4.4 Equivalence of Paths

Our equivalence checking method is based on propagating the mismatch values of the variables through all the subsequent path segments until the values match or the final path segment ending in the reset state is reached. Propagation of mismatch values from a path β_1 to the subsequent path β_2 is carried out by associating a propagated vector at the end state of the path β_1 (or equivalently, the start state of the path β_2). In Fig. 4.2(b), for example, the propagated vector at the reset state q_{00} is $\langle T, \langle c_1, c_2, t_1, t_2, t_3, a, b, c, d, e, f, g, h\rangle\rangle$.

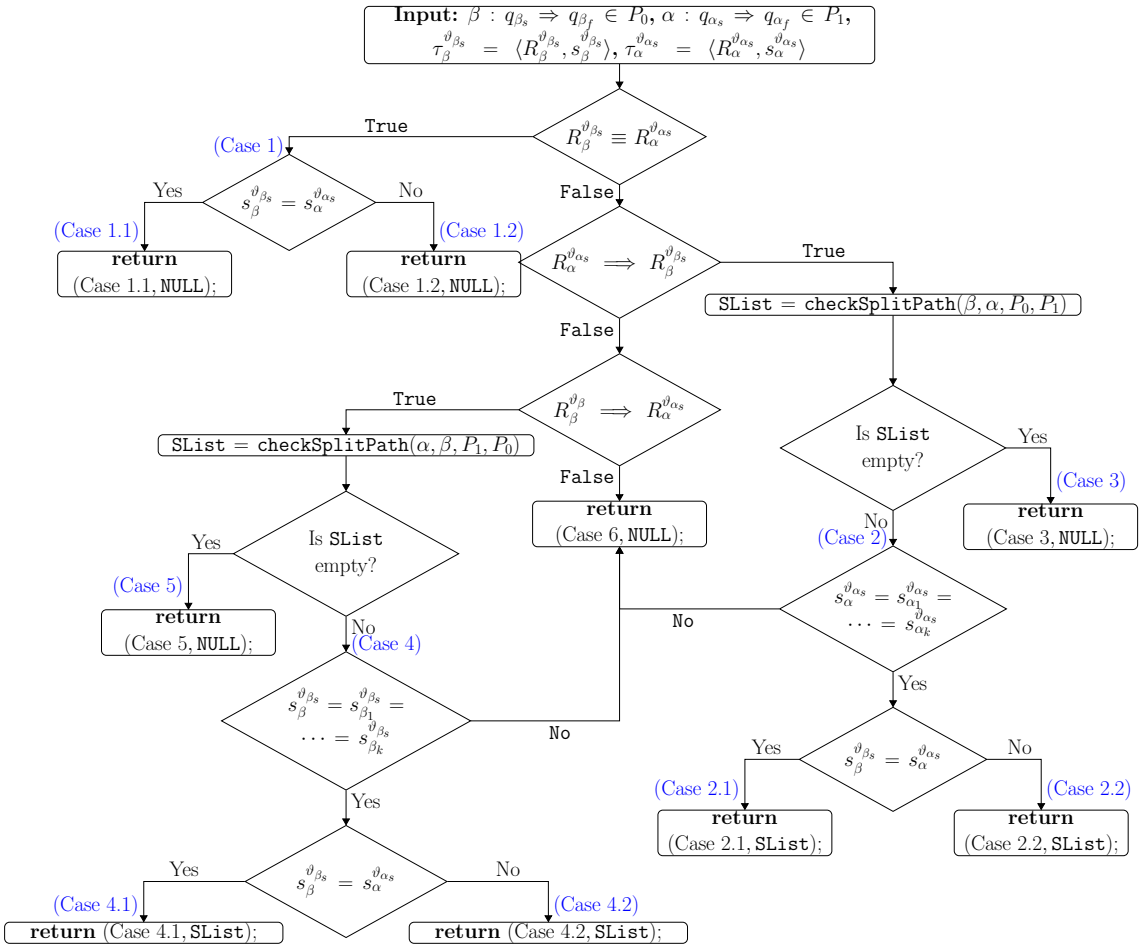
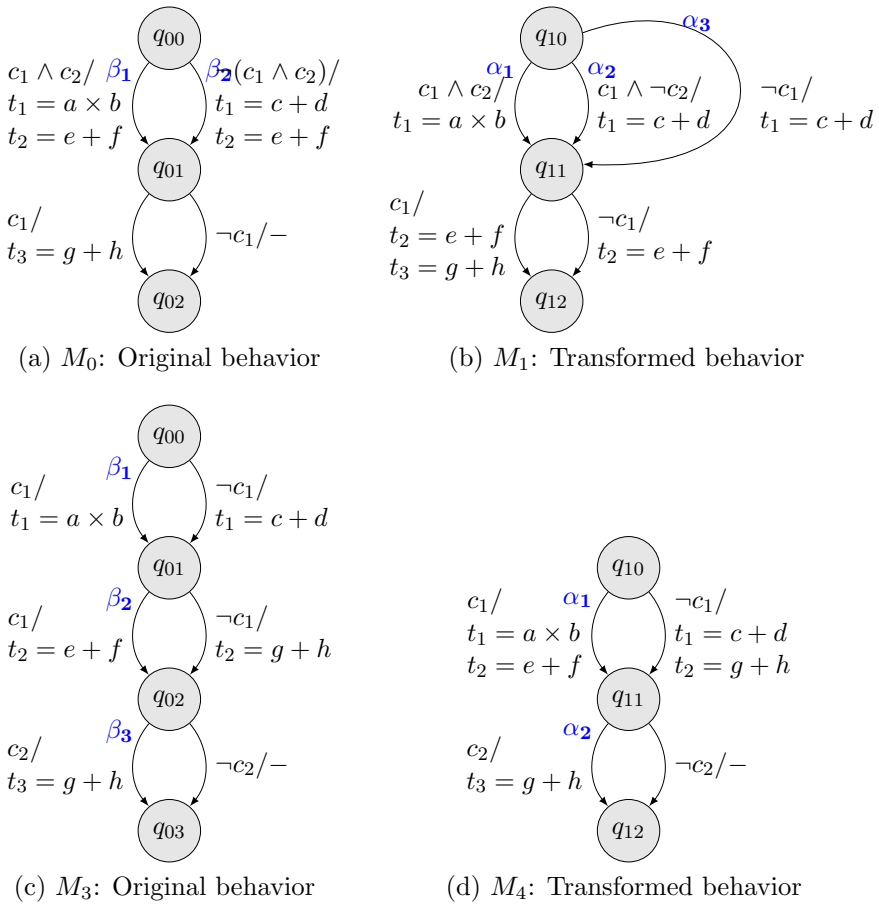


Figure 4.5: Control flow graph of `checkEquivalence`$(\beta, \alpha, \tau_\beta^{\vartheta_{\beta_s}}, \tau_\alpha^{\vartheta_{\alpha_s}})$ function.

The propagated vectors ϑ_{β_f} and ϑ_{α_f} are obtained as follow. $R'_{\beta_f} = R_\beta^{\vartheta_{\beta_s}}$ and $R'_{\alpha_f} = R_\alpha^{\vartheta_{\alpha_s}}$. The characteristic formulas $\tau_{\beta_f}^{\vartheta_{\beta_s}}$ and $\tau_{\alpha_f}^{\vartheta_{\alpha_s}}$ associated with path β and α, respectively are compared. If values of the variable v matches then value of v is reverted back to symbolic value v in both s'_{β_f} (of ϑ_{β_f}) and s'_{α_f} (of ϑ_{α_f}). If the values of v mismatches, then the actual expression of v in $s_\beta^{\vartheta_{\beta_s}}$ is copied to s'_{β_f} and the same of $s_\alpha^{\vartheta_{\alpha_s}}$ is copied to s'_{α_f}.

Example 8. *Let us consider the path* $\beta_1 = \langle q_{00} \xrightarrow{c_1 \wedge c_2} q_{01} \rangle$ *in* M_0 *(Fig. 4.6(a)) and the path* $\alpha_1 = \langle q_{10} \xrightarrow{c_1 \wedge c_2} q_{11} \rangle$ *in* M_1 *(Fig. 4.6(b)). Here,* $\vartheta_{00} = \bar{\rho}$ *and* $\vartheta_{01} = \bar{\rho}$ *and the characteristic formula for* β_1 *is* $\tau_{\beta_1}^{\vartheta_{00}} = \langle c_1 \wedge c_2, \langle c_1, c_2, a \times b, e + f, t_3, a, b, c, d, e, f, g, h \rangle \rangle$ *and for* α_1 *is* $\tau_{\alpha_1}^{\vartheta_{01}} = \langle c_1 \wedge c_2, \langle c_1, c_2, a \times b, t_2, t_3, a, b, c, d, e, f, g, h \rangle \rangle$.



(a) M_0: Original behavior (b) M_1: Transformed behavior

(c) M_3: Original behavior (d) M_4: Transformed behavior

Figure 4.6: Examples to illustrate different path equivalence cases discussed in Section 4.4

The propagated vector at q_{01} *is* $\vartheta_{01} = \langle c_1 \wedge c_2, \langle c_1, c_2, t_1, \mathbf{e} + \mathbf{f}, t_3, a, b, c, d, e, f, g, h \rangle \rangle$ *and at* q_{01} *is* $\vartheta_{11} = \langle c_1 \wedge c_2, \langle c_1, c_2, t_1, \mathbf{t_2}, t_3, a, b, c, d, e, f, g, h \rangle \rangle$.

The detailed formulation of the equivalence of paths are discussed next. We have the following cases while comparing these two paths β in M_0 and α in M_1:
Case 1 (One-to-one Equivalence) *Condition of execution matches, i.e.,* $R_\beta^{\vartheta_{\beta_s}} \equiv R_\alpha^{\vartheta_{\alpha_s}}$: The following subcases may occur.
Case 1.1 *Data transformation matches, i.e.,* $s_\beta^{\vartheta_{\beta_s}} = s_\alpha^{\vartheta_{\alpha_s}}$: The path β is said to be *unconditionally equivalent* to the path α (U-equivalent in short, denoted by $\beta \simeq_u \alpha$). In this case the propagated vector at the end state of the paths β and α will be $\bar{\rho}$.

Algorithm 6: `findEquivalentPath`$(\beta, \vartheta_{\beta_s}, q_{1j}, \vartheta_{q_{1j}}, P_0, P_1)$

Input : A path $\beta \in P_0$, the propagated vector ϑ_{β_s}, a state $q_{1j} \in M_1$, the propagated vector $\vartheta_{q_{1j}}$ associated with q_{1j}, and the path covers P_0 and P_1 of M_0 and M_1, respectively.

Output: An ordered tuple $\langle \gamma_1, \gamma_2, \vartheta_{\gamma_{1_f}}, \vartheta_{\gamma_{2_f}}, type, \texttt{SList} \rangle$ s.t. $\gamma_1 \simeq_u \gamma_2$ or $\gamma_1 \simeq_c \gamma_2$ or $\gamma_1 \simeq_{us} \gamma_2$ or $\gamma_1 \simeq_{cs} \gamma_2$, the propagated vectors $\vartheta_{\gamma_{1_f}}$ and $\vartheta_{\gamma_{2_f}}$, *type* and `SList` contains a set of paths.

1 $\tau_\beta^{\vartheta_{\beta_s}} = \langle R_\beta^{\vartheta_{\beta_s}}, s_\beta^{\vartheta_{\beta_s}} \rangle$
2 **foreach** path $\alpha : (q_{\alpha_s} \Rightarrow q_{\alpha_f}) \in P_1$ emanating from q_{1j} /* $q_{\alpha_s} = q_{1j}$ */
3 **do**
4 $\quad \tau_\alpha^{\vartheta_{\alpha_s}} = \langle R_\alpha^{\vartheta_{\alpha_s}}, s_\alpha^{\vartheta_{\alpha_s}} \rangle$
5 $\quad (type, \texttt{SList}) \leftarrow \texttt{checkEquivalence}(\beta, \alpha, \tau_\beta^{\vartheta_{\beta_s}}, \tau_\alpha^{\vartheta_{\alpha_s}})$
6 $\quad$ **if** *type* = Case 1.1 **then**
7 $\quad\quad$ **return** $(\beta, \alpha, \bar{\rho}, \bar{\rho}, type, \texttt{NULL})$;
8 $\quad$ **else if** *type* = Case 1.2 **then**
9 $\quad\quad$ **return** $(\beta, \alpha, \vartheta_{\beta_f}, \vartheta_{\alpha_f}, type, \texttt{NULL})$;
10 $\quad$ **else if** *type* = Case 2.1 **then**
11 $\quad\quad$ **return** $(\beta, \alpha, \bar{\rho}, \bar{\rho}, type, \texttt{SList})$;
12 $\quad$ **else if** *type* = Case 2.2 **then**
13 $\quad\quad \vartheta_{\alpha_f} = \langle R'_\beta, s'_\alpha \rangle$
14 $\quad\quad$ **return** $(\beta, \alpha, \vartheta_{\beta_f}, \vartheta_{\alpha_f}, type, \texttt{SList})$;
15 $\quad$ **else if** *type* = Case 3 **then**
16 $\quad\quad$ **return** $(\beta, \eta_{\alpha_s}, \vartheta_{\beta_f}, \vartheta_{\alpha_s}, type, \texttt{NULL})$;
17 $\quad$ **else if** *type* = Case 4.1 **then**
18 $\quad\quad$ **return** $(\beta, \alpha, \bar{\rho}, \bar{\rho}, type, \texttt{SList})$;
19 $\quad$ **else if** *type* = Case 4.2 **then**
20 $\quad\quad \vartheta_{\beta_f} = \langle R'_\alpha, s'_\alpha \rangle$
21 $\quad\quad$ **return** $(\beta, \alpha, \vartheta_{\beta_f}, \vartheta_{\alpha_f}, type, \texttt{SList})$;
22 $\quad$ **else if** *type* = Case 5 **then**
23 $\quad\quad$ **return** $(\eta_{\beta_s}, \alpha, \vartheta_{\beta_s}, \vartheta_{\alpha_f}, type, \texttt{NULL})$;
24 $\quad$ **else**
25 $\quad\quad$ **continue**; /* Case 6 -- $\beta \not\simeq \alpha$ */
26 $\quad$ **end if**
27 **end foreach**
/* All the paths emanating from q_{1j} are not equivalent to the path β */
28 **return** $(\beta, \Omega, \bar{\rho}, \bar{\rho}, type, \texttt{NULL})$; /* Ω denotes a non-existent path */

Example 9. *Let us consider the FSMDs M_0 and M_2 shown in Fig 4.2. The path $\beta_1 = \langle q_{00} \xrightarrow{c_1 \wedge c_2} q_{01} \rangle$ in M_0 is U-equivalent to the path $\alpha_1\alpha_3 = \langle q_{10} \xrightarrow{c_1 \wedge c_2} q_{12} \rangle$ in M_2*

, i.e., $\beta_1 \simeq_u \alpha_1\alpha_3$. *Since, there is no mismatch in the values of the variables. In this case, the propagated vector at* q_{01} *and* q_{12} *will be* $\bar{\rho}$.

Case 1.2 *Data transformation does not match, i.e.,* $s_\beta^{\vartheta_{\beta_s}} \neq s_\alpha^{\vartheta_{\alpha_s}}$: These two paths are said to be *conditionally equivalent* (C-equivalent in short, denoted by $\beta \simeq_c \alpha$) if

Condition 1 [either $q_{\beta_f} = q_{00}$ or $q_{\alpha_f} = q_{10}$ but not both]: One of the following condition is true:

- $q_{\beta_f} = q_{00}$ and there exists a concatenated path from q_{α_f} to q_{10} (say, α') such that characteristic formula for the null path at q_{00} is equal to the characteristic formula for α', i.e., $\tau_{\eta_{q00}}^{\vartheta_{00}} = \tau_{\alpha'}^{\vartheta_{\alpha_f}}$.
- $q_{\alpha_f} = q_{10}$ and there exists a concatenated path from q_{β_f} to q_{00} (say, β') $\tau_{\beta'}^{\vartheta_{\beta_f}} = \tau_{\eta_{q10}}^{\vartheta_{10}}$.

Condition 2 [$q_{\beta_f} \neq q_{00}$ and $q_{\alpha_f} \neq q_{10}$]: $\forall\beta'$ emanating from the state q_{β_f} with propagated vector $\langle R'_{\beta_f}, s'_{\beta_f}\rangle$ there exists a path α' emanating from q_{α_f} with the propagated vector $\langle R'_{\alpha_f}, s'_{\alpha_f}\rangle$, such that $\beta' \simeq_u \alpha'$ or $\beta' \simeq_{us} \alpha'$ or $\beta' \simeq_{cs} \alpha'$ or $\beta' \simeq_c \alpha'$ ($\simeq_{us}$ and $\simeq_{cs}$ are defined in case 2).

The condition 1 handles the scenario where merging of conditional blocks lead to the reset state as discussed in Subsection 4.2.3. As discussed in Section 4.5, the function `findEquivalentPathAtReset` in our proposed method handles the condition 1 and the function `ECC` invokes itself recursively to handle the condition 2.

Example 10. *In continuation to Example 8, the path* $\beta_1 = \langle q_{00} \xrightarrow{c_1 \wedge c_2} q_{01}\rangle$ *in* M_0 *(Fig. 4.6(a)) is declared as C-equivalent to the path* $\alpha_1 = \langle q_{10} \xrightarrow{c_1 \wedge c_2} q_{11}\rangle$ *in* M_1 *(Fig. 4.6(b)) i.e.,* $\beta_1 \simeq_c \alpha_1$, *since there is a mismatch in the values of the variable* t_2 *and for all the paths emanating from* q_{01} *there exists a U-equivalent path emanating from* q_{11} *with respect to propagated vectors.*

Case 2 (Split Equivalence) $R_\alpha^{\vartheta_{\alpha_s}} \implies R_\beta^{\vartheta_{\beta_s}}$ and there exists a set of paths say $A = \{\alpha_1, \ldots, \alpha_k\}$ in M_1 emanating from q_{α_s} and ending at q_{α_f} such that $R_\beta \equiv \big(R_\alpha \vee R_{\alpha_1} \vee \cdots \vee R_{\alpha_k}\big)$: This situation arises when the path β in M_0 has been split into more than one path in M_1 and the condition of execution of the path β is equivalent to the disjunction of the conditions of execution of the paths in the set $\{\alpha\} \cup A$. In this case, we have the following subcases.

Case 2.1 *Data transformation matches, i.e.,* $s_{\beta}^{\vartheta_{\beta_s}} = s_{\alpha}^{\vartheta_{\alpha_s}} = s_{\alpha_1}^{\vartheta_{\alpha_s}} = \cdots = s_{\alpha_k}^{\vartheta_{\alpha_s}}$: The path β is said to be *unconditionally split equivalent* to the path α (US-equivalent in short, denoted by $\beta \simeq_{us} \alpha$). Note that, the path β is also US-equivalent to the other paths in the set A i.e., $\beta \simeq_{us} \alpha_1, \ldots, \beta \simeq_{us} \alpha_k$.

Example 11. *Consider Fig. 4.2, the path* $\beta_2 = \langle q_{00} \xRightarrow{\neg(c_1 \wedge c_2)} q_{01} \rangle$ *in* M_0 *is US-equivalent to the paths* $\alpha_1\alpha_4 = \langle q_{10} \xRightarrow{c_1 \wedge \neg c_2} q_{12} \rangle$ *and* $\alpha_2 = \langle q_{10} \xRightarrow{\neg c_1} q_{12} \rangle$ *in* M_2 *i.e.,* $\beta_2 \simeq_{us} \alpha_1\alpha_4$ *and* $\beta_2 \simeq_{us} \alpha_2$.

Case 2.2 *Data transformation does not match, i.e.,* $s_{\alpha}^{\vartheta_{\alpha_s}} = s_{\alpha_1}^{\vartheta_{\alpha_s}} = \cdots = s_{\alpha_k}^{\vartheta_{\alpha_s}}$ but $s_{\beta}^{\vartheta_{\beta_s}} \neq s_{\alpha}^{\vartheta_{\alpha_s}}$: The path β is said to be *conditionally split equivalent* (CS-equivalent in short, denoted by $\beta \simeq_{cs} \alpha$) to the path in α if

- This condition is the same as condition 1 of Case 1.2.
- $\forall \beta'$ emanating from the state q_{β_f} with the propagated vector $\langle R'_{\beta}, s'_{\beta} \rangle$ there exists a path α' emanating from q_{α_f} with the propagated vector $\langle R'_{\beta}, s'_{\alpha} \rangle$, such that $\beta' \simeq_u \alpha'$ or $\beta' \simeq_{us} \alpha'$ or $\beta' \simeq_{cs} \alpha'$ or $\beta' \simeq_c \alpha'$.

Once the paths β and α is declared as $\beta \simeq_{cs} \alpha$, then the path β is also declared as CS-equivalent to each paths in the set A.

Example 12. *The condition of execution of the path* $\beta_2 = \langle q_{00} \xRightarrow{\neg(c_1 \wedge c_2)} q_{01} \rangle$ *in* M_0 *(Fig. 4.6(a)) is equivalent to the disjunction of condition of execution of the paths* $\alpha_2 = \langle q_{10} \xRightarrow{c_1 \wedge \neg c_2} q_{11} \rangle$ *and* $\alpha_3 = \langle q_{10} \xRightarrow{\neg c_1} q_{13} \rangle$ *in* M_1 *(Fig. 4.6(b)). The data transformations of the paths* α_2 *and* α_3 *are the same but have mismatch in the values of the variable* t_2 *with data transformations of the path* β_2*. The path* β_2 *is CS-equivalent to the paths* α_2 *and* α_3 *i.e.,* $\beta_2 \simeq_{cs} \alpha_2$ *and* $\beta_2 \simeq_{cs} \alpha_3$.

Case 3 $R_{\alpha}^{\vartheta_{\alpha_s}} \implies R_{\beta}^{\vartheta_{\beta_s}}$ and path split scenario does not arise: In this case, we introduce the null path at q_{α_s} (η_{α_s}) and the path β is declared as conditionally equivalent to η_{α_s} ($\beta \simeq_c \eta_{\alpha_s}$) if

- This condition is the same as condition 1 of Case 1.2.
- $\forall \beta'$ emanating from the state q_{β_f} with the propagated vector $\langle R'_{\beta}, s'_{\beta} \rangle$ there exists a path α' emanating from q_{α_s} with the propagated vector ϑ_{α_s}, such that $\beta' \simeq_u \alpha'$ or $\beta' \simeq_{us} \alpha'$ or $\beta' \simeq_{cs} \alpha'$ or $\beta' \simeq_c \alpha'$.

Example 13. *The path* $\beta_1 = \langle q_{00} \stackrel{c_1}{\Rightarrow} q_{01}\rangle$ *of* M_3 *in Fig. 4.6(c) is a candidate of C-equivalent to the path* $\alpha_1 = \langle q_{10} \stackrel{c_1}{\Rightarrow} q_{11}\rangle$ *of* M_4 *in Fig. 4.6(d) since there is mismatch in the values of the variable* t_2 *(Case 1.2). Therefore, the propagated vector at* q_{01} *is* $\vartheta_{01} = \langle c_1, \langle c_1, c_2, t_1, \mathbf{t_2}, t_3, a, b, c, d, e, f, g, h\rangle\rangle$ *and at* q_{11} *is* $\vartheta_{11} = \langle c_1, \langle c_1, c_2, t_1, \mathbf{e}+\mathbf{f}, t_3, a, b, c, d, e, f, g, h\rangle\rangle$*. The paths* $\beta_2 = \langle q_{01} \stackrel{c_1}{\Rightarrow} q_{02}\rangle$ *and* $\alpha_2 = \langle q_{11} \stackrel{c_2}{\Rightarrow} q_{12}\rangle$ *are compared next. The* $R_{\beta_2}^{\vartheta_{01}}$ *is* c_1 *with respect to* ϑ_{01} *and* $R_{\alpha_2}^{\vartheta_{11}}$ *is* $c_1 \wedge c_2$ *with respect to* ϑ_{11}*. Since,* $R_{\alpha_2}^{\vartheta_{11}} \implies R_{\beta_2}^{\vartheta_{01}}$*, we should check for the path merge/split scenario (i.e., Case 2). However, Case 2 does not matches here. Therefore the path* β_2 *is compared with the null path* $\eta_{q_{11}}$ *and declared as* $\beta_2 \simeq_c \eta_{q_{11}}$ *(Case 3). Now the propagated vector at* q_{02} *is* $\vartheta_{02} = \langle c_1, \langle c_1, c_2, t_1, \mathbf{e}+\mathbf{f}, t_3, a, b, c, d, e, f, g, h\rangle\rangle$*. The path* $\beta_3 = \langle q_{02} \stackrel{c_2}{\Rightarrow} q_{03}\rangle$ *and the path* α_2 *are compared next. The path* β_3 *is declared as U-equivalent to the path* α_2 *with respect to propagated vector associated with their respective start state (Case 1.1). In the similar way other paths are compared and at the end of the execution of equivalence checking both the behaviors are declared equivalent.*

In the following, cases 4 and 5 are the reverse situation of the cases 2 and 3, respectively.

Case 4 $R_{\beta}^{\vartheta_{\beta_s}} \implies R_{\alpha}^{\vartheta_{\alpha_s}}$*, and* there exists a set of paths say $B = \{\beta_1, \ldots, \beta_k\}$ in M_0 emanating from q_{β_s} and ending at q_{β_f} such that $\left(R_\beta \vee R_{\beta_1} \vee \cdots \vee R_{\beta_k}\right) \equiv R_\alpha$: The condition of execution of the path α in M_1 is equivalent to the disjunction of the conditions of execution of each path in the set $\{\beta\} \cup B$ in M_0. In this case we have the following subcases.

Case 4.1 $s_{\beta}^{\vartheta_{\beta_s}} = s_{\beta_1}^{\vartheta_{\beta_s}} = \cdots = s_{\beta_k}^{\vartheta_{\beta_s}} = s_{\alpha}^{\vartheta_{\alpha_s}}$: The each path in the set $\{\beta\} \cup B$ is US-equivalent to the path α i.e., $\beta \simeq_{us} \alpha, \beta_1 \simeq_{us} \alpha, \ldots, \beta_k \simeq_{us} \alpha$.

Case 4.2 $s_{\beta}^{\vartheta_{\beta_s}} = s_{\beta_1}^{\vartheta_{\beta_s}} = \cdots = s_{\beta_k}^{\vartheta_{\beta_s}}$ but $s_{\alpha}^{\vartheta_{\alpha_s}} \neq s_{\beta}^{\vartheta_{\beta_s}}$: The path β is conditionally split equivalent to the path α if

- This condition is the same as condition 1 of Case 1.2.
- $\forall \beta'$ emanating from the state q_{β_f} with the propagated vector $\langle R'_\alpha, s'_\beta\rangle$ there exists a path α' emanating from q_{α_f} with the propagated vector $\langle R'_\alpha, s'_\alpha\rangle$, such that $\beta' \simeq_u \alpha'$ or $\beta' \simeq_{us} \alpha'$ or or $\beta' \simeq_{cs} \alpha'$ or $\beta' \simeq_c \alpha'$.

Case 5 $R_{\beta}^{\vartheta_{\beta_s}} \implies R_{\alpha}^{\vartheta_{\alpha_s}}$*, but path split scenario does not arise:* The null path at q_{β_s} is declared as conditionally equivalent to the path α, i.e., $\eta_{\beta_s} \simeq_c \alpha$ if

- This condition is the same as condition 1 of Case 1.2.

- for the $\forall\beta$ emanating from the state q_{β_s} with the propagated vector ϑ_{β_s} there exists a path α' emanating from q_{α_f} with propagated vector $\langle R'_\alpha, s'_\alpha\rangle$, such that $\beta' \simeq_u \alpha'$ or $\beta' \simeq_{us} \alpha'$ or or $\beta' \simeq_{cs} \alpha'$ or $\beta' \simeq_c \alpha'$.

Case 6 The path β and α is not equivalent if one of the condition among the following conditions is true:

- $R_\beta^{\vartheta_{\beta_s}} \not\equiv R_\alpha^{\vartheta_{\alpha_s}}$ and $R_\beta^{\vartheta_{\beta_s}} \not\Longrightarrow R_\alpha^{\vartheta_{\alpha_s}}$ and $R_\alpha^{\vartheta_{\alpha_s}} \not\Longrightarrow R_\beta^{\vartheta_{\beta_s}}$.

- In the case 2 or case 4, data transformation of the paths in the set A or B are not the same.

To check the equivalence, we use the function `findEquivalentPath` (Algorithm 6) which invokes the function `checkEquivalence` (Fig. 4.5) to find an applicable case. The control flow of the function `checkEquivalence` is given in Fig. 4.5. This function takes as input two paths (β and α) and the characteristics formula associated with these paths. It finds the equivalence relationship between the paths β and α and returns the one of the type of the case defined above under which these two paths have an equivalence relationship. It invokes the function `checkSplitPath`$(\beta, \alpha, P_0, P_1)$ which checks whether the path $\beta \in P_0$ has been split more than one path presents in P_1. The $\alpha \in P_1$ is one path among the split paths. If a path has been split then the function `checkSplitPath` returns a list `SList` which contains a set of paths which satisfy the condition mentioned in Case 2 or Case 4.

Algorithm 7: `containmentChecker`(M_0, M_1)

```
1 Compute the path cover P_0 and P_1 of M_0, M_1, respectively; W_csp = (q_00, q_10);
    Set E_u, E_c and LIST as empty set;
2 foreach (q_0i, q_1j) ∈ W_csp do
3 |   if ECC (M_0, M_1, q_0i, q_1j, P_0, P_1, W_csp, E_u, E_c, LIST) returns "failure" then
4 |   |   Report "unable to decide M_0 ⊑ M_1" and exit;
5 |   end if
6 end foreach
7 Report "M_0 ⊑ M_1";
```

Algorithm 8: ECC($M_0, M_1, q_{0i}, q_{1j}, P_0, P_1, W_{csp}, E_u, E_c, LIST$)

foreach path $\beta : (q_{0i} \Rightarrow q_{0m})$ in P_0 **do**
  if $R'_{0i} \wedge R_{\beta}^{\vartheta_{0i}} \not\equiv$ `False` **then**
    if q_{0i} is a loop header and `checkFalseComputation`(q_{0i}) returns `True` **then**
      continue;
    end if
    if Path β is already present in the $LIST$ **then**
      continue; `/* prevent recursions */`
    end if
    $(\beta', \alpha, \bar{\vartheta}_{\beta'_f}, \bar{\vartheta}_{\alpha_f}, type,$ `SList`$) \leftarrow$ `findEquivalentPath`$(\beta, \vartheta_{0i}, q_{1j}, \vartheta_{1j}, P_0, P_1)$;`/*` $\alpha : (q_{1j} \Rightarrow q_{1n})$ `*/`
    if *type* is Case 1.1 or Case 2.1 or Case 4.1 **then**
      $E_u = E_u \cup \{(\beta', \alpha)\}$;
      if `SList` is not empty **then**
        $E_u = E_u \cup \{(\beta', \gamma_i)\}$, $\forall \gamma_i \in$ `SList`; for Case 2.1
        $E_u = E_u \cup \{(\gamma_i, \alpha)\}$, $\forall \gamma_i \in$ `SList`; for Case 4.1
      end if
      $W_{csp} = W_{csp} \cup \{(q_{0m}, q_{1n})\}$;
    else if *type* is Case 1.2 or Case 2.2 or Case 3 or Case 4.2 or Case 5 **then**
      if $q_{\beta'_f}$ and q_{α_f} are the reset state **then**
        return *failure*;
      else if $q_{\beta'_f}$ or q_{α_f} is the reset state **then**
        if !`findEquivalentPathAtReset`$(M_0, M_1, \beta', \alpha,$
        $\tau_{\beta'}^{\vartheta_{0i}}, \tau_{\alpha}^{\vartheta_{1j}}, P_0, P_1, E_u, E_c)$ **then**
          return *failure*;
        else
          $E_c = E_c \cup \{(\beta', \alpha)\}$;
          if `SList` is not empty **then**
            $E_c = E_c \cup \{(\beta', \gamma_i)\}$, $\forall \gamma_i \in$ `SList`; for Case 2.2
            $E_c = E_c \cup \{(\gamma_i, \alpha)\}$, $\forall \gamma_i \in$ `SList`; for Case 4.2
          end if
          continue ;
        end if
      else if $q_{\beta'_f}$ or q_{α_f} appears as the end state of some path already in
       $LIST \wedge$ `loopInvariant`$(\beta', \alpha, \bar{\vartheta}_{\beta'_f}, \bar{\vartheta}_{\alpha_f})$ **then**
        return *failure*; `/* not loop invariant */`
      else
        $\vartheta_{\beta'_f} \leftarrow \bar{\vartheta}_{\beta'_f}$; $\vartheta_{\alpha_f} \leftarrow \bar{\vartheta}_{\alpha_f}$;
        Append $\langle \beta', \alpha \rangle$ to $LIST$
        if `SList` is not empty **then**
          $LIST = LIST \cup \{(\beta', \gamma_i)\}$, $\forall \gamma_i \in$ `SList`; for Case 2.2
          $LIST = LIST \cup \{(\gamma_i, \alpha)\}$, $\forall \gamma_i \in$ `SList`; for Case 4.2
        `ECC`$(M_0, M_1, q_{\beta'_f}, q_{\alpha_f}, P_0, P_1, W_{csp}, E_u, E_c, LIST)$;
      end if
    else
      return *failure*; `/* Fail to find the path` α `*/`
    end if
  end if
end foreach
$E_c = E_c \cup \{$Last member of $LIST\}$;
If the $LIST$ contains some other path pairs whose start(end) state is the same as last member start(end) state then append these path pairs in the set E_c and remove them from the $LIST$
$LIST \leftarrow LIST \backslash \{$Last member of $LIST\}$;
return *success*;

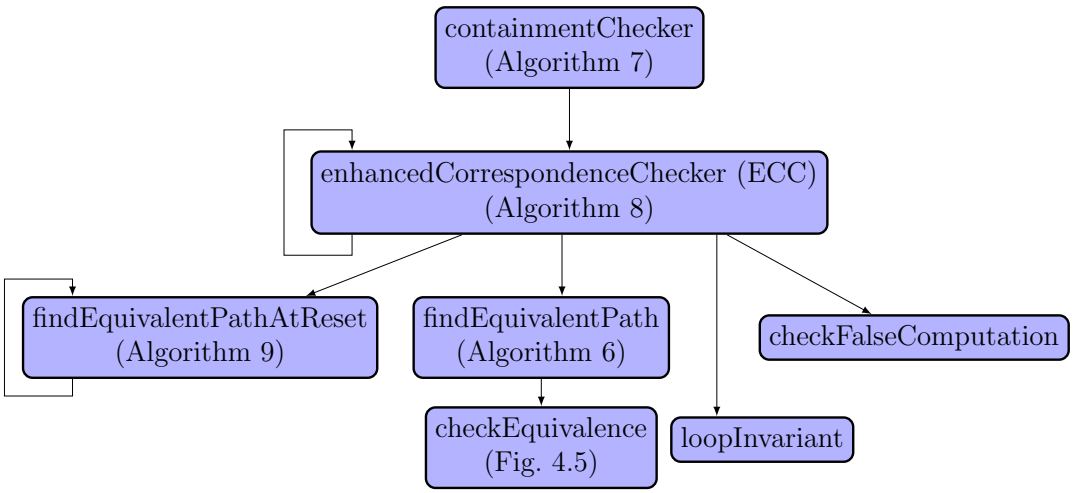


Figure 4.7: A overall flow of our verification method

4.5 Overall Verification Method

The overall flow of our verification method is given in Fig. 4.7. We begin the procedure of equivalence checking by invoking the function `containmentChecker` (Algorithm 7). This function first identifies the cutpoints based on scheme S_3 as discussed in Subsection 4.3.2 in each FSMD, followed by computing their path covers. It also initializes W_{csp}, a set of corresponding state pairs by inserting the reset state pair (q_{00}, q_{10}). The sets E_u and E_c contain the U(S)-equivalent and C(S)-equivalent path pairs, respectively. The function `containmentChecker` invokes an enhanced correspondence checker (`ECC`) function (Algorithm 8) for each corresponding state pairs (in step 3) [4]. Depending on the output returned by `ECC`, `containmentChecker` outputs the decision whether the original FSMD is contained in the transformed FSMD or not. The *LIST* is used to store the candidate of C(S)-equivalent path pairs visited along the chain of recursive invocation of `ECC`

[4]It may be noted that the `ECC` function presented in this Chapter is an enhanced version of the `ECC` function presented in Chapter 3. We keep the name of this function same as of Chapter 3 to keep the consistency among two equivalence checkers presented in two Chapters. Similarly, the function `findEquivalentPath` of this Chapter is an enhanced version of the same function of Chapter 3. Further, we are using the same `checkFalseComputation` and `loopInvariant` functions of Chapter 3 here.

Algorithm 9: `findEquivalentPathAtReset`$(M_0, M_1, \beta, \alpha, \tau_\beta, \tau_\alpha, P_0, P_1, E_u, E_c)$

```
1  if q_{β_f} is the reset state in M_0 then
2      if q_{α_f} is the reset state in M_1 then
3          if τ_β = τ_α then
4              E_u = E_u ∪ {(η_{β_f}, α)};
5              return success;
6          else
7              return failure;
8          end if
9      else if There exists a path γ in P_1 emanating from q_{α_f} such that
           R_β ≡ R_γ^{ϑ_{α_f}} then
10         E_c = E_c ∪ {(η_{β_f}, γ)};
11         findEquivalentPathAtReset(M_0, M_1, β, γ,
12         τ_β, τ_γ^{ϑ_{α_f}}, P_0, P_1, E_u, E_c);
13     else
14         return failure;
15     end if
16 else
17     if q_{β_f} is the reset state in M_0 then
18         if τ_β = τ_α then
19             E_u = E_u ∪ {(β, η_{α_f})};
20             return success;
21         else
22             return failure;
23         end if
24     else if There exists a path γ in P_0 emanating from q_{β_f} such that
           R_α ≡ R_γ^{ϑ_{β_f}} then
25         E_c = E_c ∪ {(γ, η_{α_f})};
26         findEquivalentPathAtReset(M_0, M_1, γ, α,
27         τ_γ^{ϑ_{β_f}}, τ_α, P_0, P_1, , E_u, E_c);
28     else
29         return failure;
30     end if
31 end if
```

invoked by `containmentChecker`.

The function `ECC` (Algorithm 8) is the key function of our verification method. The function takes as input two FSMDs M_0 and M_1, a corresponding state pair

(q_{0i}, q_{1j}), path covers P_0 (of M_0) and P_1 (of M_1), a corresponding state pair set W_{csp}, $LIST$ which maintains a candidate of C(S)-equivalent pairs of paths and E_u and E_c for storing the U(S)-equivalent and C(S)-equivalent path pairs, respectively. The function `ECC` returns "success" if for every path emanating from q_{0i} an equivalent path originating from q_{1j} is found (in step 50 of Algorithm 8); otherwise, it returns "failure". To avoid the false computation [56] at a loop header `ECC` invokes the function `checkFalseComputation` (in step 3). The function `checkFalseComputation` returns `False` if the loop at q_{0i} under the propagated condition will execute at least once over all possible inputs in M_0. It returns `True` otherwise. The function `ECC` invokes the function `findEquivalentPath` (Algorithm 6) to find a U(S)- or C(S)-equivalent path $\alpha : (q_{1j} \Rightarrow q_{1n})$ in the transformed FSMD M_1 for each path $\beta : (q_{0i} \Rightarrow q_{0m})$ starting from state q_{0i} of the original FSMD M_0. The function `findEquivalentPath` returns $(\beta', \alpha, \bar{\vartheta}_{\beta'_f}, \bar{\vartheta}_{\alpha_f}, type, \texttt{SList})$, where $\beta' = \beta$ for all values of the variable *type* except when the variable *type* is Case 3. When value of the variable *type* is Case 3 then β' is defined as a null path at the starting state of β. When the variable *type* is Case 6 then α does not exist and `findEquivalentPath` returns a non-existent path Ω in place of α (i.e., M_0 and M_1 may not be equivalent, handled in step 43) , otherwise, α is an equivalent path to β. $\bar{\vartheta}_{\beta'_f}$ and $\bar{\vartheta}_{\alpha_f}$ are to be propagated to end state of β' and α, respectively. An equivalence relationship between the path β and α is defined by the variable *type*. The list `SList` contains a set of path pairs when *type* is Case 2 or Case 4.

If the function `findEquivalentPath` fails to find a path α such that $\beta \simeq_u \alpha$ or $\beta \simeq_{us} \alpha$ or $\beta \simeq_c \alpha$ or $\beta \simeq_{cs} \alpha$ (i.e., α does not exists), then this cause `ECC` to return "failure" as shown in step 43. If the function `findEquivalentPath` finds a path α such that $\beta \simeq_u \alpha$ or $\beta \simeq_{us} \alpha$, then the function `ECC` inserts the path pair (β, α) into E_u (in step 11) and if the `SList` is not empty then updates the E_u as shown in steps 12 and 14; `ECC` also declares the end state of β and α as a corresponding state pair and insert this state pair into W_{csp} (in step 16). If the function `findEquivalentPath` finds a path α such that $\beta' \simeq_c \alpha$ or $\beta' \simeq_{cs} \alpha$ then further value propagation is required. However, we need to check the following scenarios first:

1. Since the computation cannot extend beyond the reset state; therefore, if the end states of β' and α are the reset states then `ECC` returns "failure" as shown in step 19;

2. If the end state of β' is the reset state and the end state of α is not the reset state then it may possible that from the end state to the reset state of α, there exists an unique set of concatenated paths which is unconditionally equivalent to null path at the reset state of β (vice versa). To find such path, `ECC` invokes the function `findEquivalentPathAtReset` (in step 21) this function returns `True` if such a concatenated path exists; otherwise, returns `False`. It may be noted that these steps is required to handle the scenario presented in Subsection 4.3.3. If the function `findEquivalentPathAtReset` returns `True` then `ECC` inserts the path pair (β, α) into E_c (in step 25) and if the `SList` is not empty then updates the E_c as shown in steps 27 and 28. The function `ECC` moves back to step 1 to find an equivalence of other paths emanating from the state q_{0i} as shown in step 30.
3. If a loop has been crossed over then the `ECC` invokes the function `loopInvariant`. The function `loopInvariant` checks for the loop invariance of the propagated vector $\bar{\vartheta}_{\beta'_f}$ and $\bar{\vartheta}_{\alpha_f}$. If `loopInvariant` returns `False` then the `ECC` returns "failure" (in step 33). Note that this function is required to check the validity of code motion involving loops. The details of this function can be found in [56].

If both the checks at step 18 and step 32 resolve success then the propagated vector of the final state of the paths β' and α are updated (in step 35) and `ECC` calls itself recursively (in step 40). When `ECC` reaches step 47, it implies that for every paths emanating from the state q_{0i}, there exists a corresponding paths emanating from q_{1j} such that their final paths are U(S)-equivalent.

4.6 Correctness of the Equivalence Checking Procedure

4.6.1 Correctness

Lemma 3. *If the Algorithm 7 terminates successfully at step 7 then each path p_{0i} in path cover P_0 of M_0 is either unconditionally equivalent or unconditionally split equivalent or conditionally equivalent or conditionally split equivalent to null path or some path in path cover P_1 of M_1.*

Proof. The steps 11, 12, 25, 26, 47 of `ECC` function and the steps 4, 10, 19, 25 of `findEquivalentPathAtReset` function ensure that for each path in M_0 there is

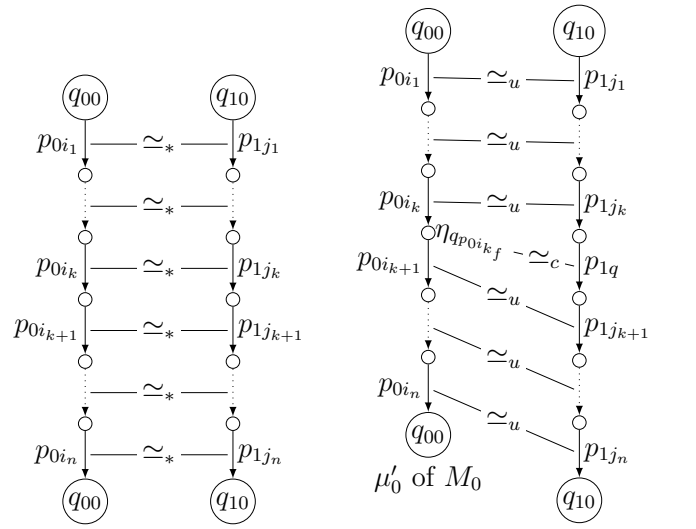


Figure 4.8: A scenario (a) where each path in path cover of M_0 has an U(S)equivalent or C(S) equivalent path in M_1 ($\simeq_*$ is $\simeq_u$ or $\simeq_{us}$ or $\simeq_c$ or $\simeq_{cs}$); (b) where a null path in M_0 has an C-equivalent path in M_1

an U(S)-equivalent or C(S)-equivalent path exists in another FSMD M_1. ■

Lemma 4. *If the Algorithm 7 terminates successfully at step 7 then each path p_{0i} leading to the reset state in path cover P_0 of M_0 will satisfy one of the condition:*

1. *Path p_{0i} is unconditionally (split) equivalent to a path p_{1j} in the path cover P_1 of M_1. Path p_{1j} also leads to the reset state of M_1.*
2. *Path p_{0i} is conditionally equivalent the path p_{1j} in the path coverP_1 of M_1. Path p_{1j} does no lead to reset state. The null path at the reset state of M_0 will have unconditionally equivalent path in P_1 leading to the reset state of M_1.*

Proof. The steps 18, 21 (invoking the function `findEquivalentPathAtReset`) and 23 of the function `ECC` ensures that condition 1 and 2 must be satisfied. ■

Theorem 4 (Correctness of the approach)**.** *If the Algorithm 7 terminates successfully at step 7 then for each computation of M_0 there exist an equivalent computation in M_1.*

Proof. Let us consider a computation μ_0 of M_0 is $\mu_0 = [p_{0i_1}, p_{0i_2}, \ldots, p_{0i_n}]$, where $p_{0i_k} \in P_0, 1 \leq k \leq n$, where $p_{0i_{1f}} = p_{0i_{nf}} = q_{00}$. From Lemma 3, there exists a sequence S of paths $\mu_1 = [p_{1j_1}, p_{1j_2}, \ldots, p_{1j_n}]$ where $p_{1j_k} \in P_1, 1 \leq k \leq n$, such that $p_{0i_k} \simeq_u p_{1j_k}$ or $p_{0i_k} \simeq_{us} p_{1j_k}$ or $p_{0i_k} \simeq_c p_{1j_k}$ or $p_{0i_k} \simeq_{cs} p_{1j_k}$.

The sequence S represents a computation if it is the concatenation of consecutive paths in M_1 starting and ending back to the reset state. The sequence S represents a computation of M_1 if μ_0 does not involve any null path. This scenario is depicted in Fig. 4.8(a). If μ_0 involves null path, then the sequence of paths in S will not be consecutive. This situation is depicted in Fig. 4.8(b). In this case, the path p_{1q}, which is not the part of S, starting from the end state of $p_{1j_{kf}}$ and ending to the start state of $p_{1j_{k+1_s}}$ is conditionally equivalent to null path at the state $q_{p_{0i_{kf}}}$ in M_0. Hence, the sequence S is not consecutive. Let $\mu_0' = [p_{0i_1}, p_{0i_2}, \ldots, \eta_{q_{p_{0i_{kf}}}}, \ldots, p_{0i_n}]$ be a computation obtained from μ_0 by introducing null paths at $q_{p_{0i_{kf}}}$ which have (un)conditionally equivalent path p_{1q} in M_1.

The Lemma 4 implies that whatever mismatches present in any path in μ_0' must get resolved when the last paths are traversed or by introducing the null path at the reset state. In this case sequence S' of paths $\mu_1 = [p_{1j_1}, p_{1j_2}, \ldots, p_{1q}, \ldots, p_{1j_n}]$ corresponding to μ_0' represents a concatenated path that starts and ends at the reset state q_{10}. Let the sequence S' represents the computation μ_1 of M_1. Thus computations μ_1 and μ_0' are equivalent. Since introducing the null path does not alter the computation μ_0 hence computations μ_1 and μ_0 are also equivalent.

Note that, if a path p_{0i_k} is (un)conditionally split equivalent to a set of the paths in M_1 then the computation μ_0 will be equivalent to each computation corresponding to each path in the set. It may be noted that for a given input, the computation μ_0 will be equivalent to exactly one of the computation since the condition of execution of the paths present in the set are mutually exclusive and the data transformation of these paths are the same. ■

Theorem 5 (Partial correctness)**.** *If the Algorithm 7 terminates successfully at step 7 then* $M_0 \sqsubseteq M_1$.

Proof. If the verification method terminates at step 7 of the function `containmentChecker` then we need to prove the following claims.

1. The set $E = E_u \cup E_c$ contains a member for each path in the path cover P_0.

2. All the paths of P_0 leading to the reset state will have unconditionally (split) equivalent path in P_1 leading to the reset state of M_1 or in the case of conditionally equivalent path, the null path at the reset state will have unconditionally equivalent path in P_1 leading to the reset state of M_1.

The set E_u contains the pair of U(S)-equivalent paths, and the set E_c contains the pair of C(S)-equivalent paths. A pair of paths is added to the set E_u and E_c at steps 11, 12, 25, 26, 47 of `ECC` function and at steps 4, 10, 19, 25 of `findEquivalentPathAtReset` function. The pair of paths declared by `findEquivallentPath` is actually U(S)-equivalent or C(S)-equivalent. The fact that $E = E_u \cup E_c$ contains a member for each path in the path cover P_0 can be proved in a way similar to presented in [42].

Claim 2 can be proved by contradiction. Let the paths $\beta \in P_0$ or $\alpha \in P_1$ lead to the reset states of M_0 and M_1, respectively and $\beta \simeq_c \alpha$. In this case, the function `ECC` returns *failure* to `containmentChecker` as shown at step 19 or step 23; consequently, `containmentChecker` terminates at step 4, not at step 7, is a contradiction. ■

4.6.2 Termination

Theorem 6 (Termination)**.** *Our verification method always terminates.*

Proof. The Fig. 4.7 represents the overall flow of our verification method. In the following, we show that each of the function involved in Fig. 4.7 terminates. In `checkFalseComputation`, we check certain property therefore it always terminates. The function `loopInvariant` always terminates since it involves a comparison of two propagated vectors. The function `findEquivalentPath` (Algorithm 6) tries to find a path α starting from $q_{1j} \in M_1$ by invoking the function `checkEquivalence`. In worst case, it checks all the paths of P_1 starting from q_{1j} which is finite. Hence it terminates as well. The function `findEquivalentPathAtReset` (Algorithm 9) finds a concatenated path from a state (say q_{ij}) to the reset state of one FSMD such that it is U-equivalent to null path at reset state in other FSMD. If the function fails to find such a concatenated path then it terminates at step 7 or 22. The function `findEquivalentPathAtReset` invokes itself recursively with the end state of path α (or β) until reset state has not been reached. This function can

only be in infinite recursion if there is a loop in the path for q_{ij} to the reset state. However, the condition at steps 9 and 24 ensure that this function can not traverse a loop twice. Therefore, the function invokes `findEquivalentPathAtReset` itself recursively only a finite number of times and always terminates.

In Algorithm 8, the outermost loop (step 1–step 46) of the function `ECC` is executed only $|P_0|$ (number of elements in P_0) time which is finite. In Algorithm 8 `ECC` can invoke itself recursively. The `ECC`(q_{0i}, q_{1j}) invokes itself with the end state of some path β emanating from q_{0i} and some path α emanating from q_{1j}. If the end state of β and α is a reset state or the function `findEquivalentPathAtReset` return `Flase` then `ECC`(q_{0i}, q_{1j}) returns *failure*. Since the recursive call of the function `ECC` does not extend beyond the reset state and the function `ECC` avoids traversing the loop twice (at step 6); therefore, the function `ECC` invokes itself recursively only a finite number of times. Hence, the Algorithm 8 also terminates.

Finally let us consider the Algorithm 7. The loop in `containmentChecker` depends upon the size of W_{csp} (a set of corresponding state pairs). Since the number of states in both the FSMDs is finite, the number of elements in W_{csp} has to be finite. So, the Algorithm 7 will also terminates. Hence our verification method always terminates. ■

4.6.3 Complexity

Let assume that there are n states in the FSMDs, and k is the maximum number of parallel edges between any two states and x is time taken to check the equivalence of the two formulas. The complexity of our method is in the product of the following three terms:

1. The first term is the complexity of `findEquivalentPath`$(\beta, q_{1j}, \cdots)$.
2. The second term is the number of times ECC is called from `containmentChecker`.
3. The third term is the number of times `ECC` calls itself recursively.

There are two cases where the worst case may arise in our method. In first case the worst case scenario arises when the cutpoint selection scheme S_3 selects all the states as cutpoint and the mismatch of paths resolves at the last recursion of `ECC`. The function `findEquivalentPath`$(\beta, q_{1j}, \cdots)$ checks all the paths from

q_{1j} in worst case. In our method, `findEquivalentPath` also checks that whether a path has been split or not. Note that to check the path merge/split scenario, it is sufficient to visit all the paths emanating from a state once. Thus, the overall the complexity of `findEquivalentPath` is $O(k \cdot n \cdot x)$. The number of times `ECC` is called from `containmentChecker` is the same as the size of the set of corresponding states pairs which is $O(n)$. In this worst case `ECC` can recursively call itself $k \cdot (n-1) + k^2 \cdot (n-1) \cdot (n-2) + \cdots + k^{n-1} \cdot (n-1) \cdot (n-2) \ldots 2 \cdot 1 \simeq k^{n-1} \cdot (n-1)^{n-1}$ times. Therefore, the complexity of overall verification method is $O((x \cdot k \cdot n) \cdot (n) \cdot (k^{n-1} \cdot (n-1)^{n-1})) \simeq O(x \cdot k^n \cdot n^{n+1})$ in this worst case.

The second worst case scenario arises when all the states except the reset state belongs the internal states of an `if-else` block. So, S_3 selects only reset state as a cutpoint. Therefore, the number of paths emanating from the reset state is $(k \cdot n)^n$. Therefore the complexity of the `findEquivalentPath` is $O((k \cdot n)^n \cdot x)$. The complexity of the other two terms is $O(1)$ since `containmentChecker` calls `ECC` only one time and there will be no recursive call of `ECC`. Hence, the overall complexity of our method is $O((k \cdot n)^n \cdot x)$ in this worst case.

From the above discussion it is clear that the complexity of our method is $O(x \cdot k^n \cdot n^{n+1})$. If we ignore the time taken by the SMT solver Z3 then the worst case complexity of the our method is the same as that of the VP and the EVP method.

4.7 Experimental Results

Our verification method discussed in Section 4.5 has been implemented in C. All the experiments have been conducted on a laptop with 1.8 GHz Intel i5 processor with 8 GB of RAM. We take the codebase of the EVP method and implement our method on the top of this codebase. To check the equivalence of the condition of executions, we replace the normalization technique [101] by the SMT-based technique in our equivalence checking framework [5]. Specifically, we use Z3 SMT solver [57] for this purpose. This is done to avoid the limitations of the normalization technique. Specifically, the checking the path merge/split scenario is not

[5]We use the EVP with this modification for experimentation here. Therefore, the run time of the EVP reported in Chapter 3 for a benchmark will not be the same here. However, this change removes the limitation of normalization technique from our equivalence checking framework.

Table 4.2: Experimental results on the benchmarks presented in [42]

Benchmarks	EVP							Our							Runtime (ms)	
	#Cut		#Path		#FEP	Equi	Exp.	#Cut		#Path		#FEP	Equi	Exp.	EVP	Our
	M_0	M_1	M_0	M_1				M_0	M_1	M_0	M_1					
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)
TLC	10	6	20	16	20	Eq	75	6	6	16	16	16	Eq	65	143	107
DIFFEQ	3	3	3	3	3	Eq	29	3	3	3	3	3	Eq	29	28	28
GCD	7	4	11	8	13	Eq	125	4	4	8	8	8	Eq	73	121	48
PERFECT	5	4	7	6	7	Eq	73	4	3	6	5	6	Eq	73	34	37
MODN	6	6	9	9	9	Eq	29	6	6	9	9	9	Eq	29	40	40
LRU	21	20	39	38	39	Eq	73	11	10	32	31	27	Eq	139	199	247
DHRC	15	13	27	24	14	Eq	135	13	12	26	24	14	Eq	177	281	241
BARCODE	28	24	55	57	75	Eq	189	13	13	54	54	46	Eq	213	657	500

#cut: Number of cutpoints in an FSMD.
#Path: Number of paths in an FSMD.
#FEP: Number of times `findEquivalentPath` function is called.
Exp.: Maximum length of a formula in terms of variables along with that of operations.
Eq: M_0 and M_1 are equivalent.

feasible with the normalization technique. We run both the EVP and our method with Z3 for checking condition of execution for fair evaluation of runtime. However, we still use the normalization technique to check the equivalence of data transformations.

In our first experiment, all the benchmarks listed in Table 4.2 are taken from [42]. The benchmarks TLC and GCD are control-intensive, the benchmarks DIFFEQ, PERFECT, and MODN are data-intensive, and the benchmarks LRU, DHRC and BARCODE are both control and data-intensive. The transformed FSMD is obtained from the original one in two steps. First, we obtained the intermediate transformed FSMD by running the SPARK tool [15] on these benchmarks. We forced SPARK to apply the code transformation like copy and constant propagation, common sub-expression elimination, dead code elimination (DCE) and loop invariant code motion to the original behavior to produce the corresponding optimized transformed behavior. The intermediate transformed FSMD obtained by SPARK is converted into the final transformed FSMD according to path-based

Table 4.3: Experimental results on the benchmarks presented in [44, 104, 105], CHStone benchmarks [55] and the benchmarks listed in Bambu HLS tool [14]

Benchmarks			EVP				Our							
		#C	#Path		Equi-	T	#Path		Equi-	#PSE	#Exp.	#SMT	SMT_Time	T
			M0	M1	valent	(ms)	M0	M1	valent		length		(ms)	(ms)
(1)		(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)
Fig. 4.4		12	5	3	MNEq	6	5	3	Eq	0	34	7	2	16
Fig. 4.2 [44]		18	6	8	MNEq	9	6	7	Eq	1	74	27	7	25
Peñalba [44]		34	14	15	MNEq	26	13	14	Eq	1	89	55	18	65
Juan [104]		20	8	9	MNEq	12	7	8	Eq	1	86	34	10	36
Jian [105]		16	10	11	MNEq	15	8	9	Eq	1	90	38	12	56
Bambu	WAKA	35	4	3	Eq	77	4	3	Eq	0	83	14	6	77
	ARF	43	5	5	Eq	357	5	5	Eq	0	96	51	76	357
	MOTION	44	1	1	Eq	72	1	1	Eq	0	0	0	0	72
CHStone	BLOWFISH	151	21	21	Eq	159	19	19	Eq	0	50	66	28	114
	GSM	240	96	86	Eq	1651	76	76	Eq	0	154	348	956	1413
	MIPS	259	77	51	MNEq	105	45	45	Eq	2	491	779	4378	5894
	AES	330	132	96	MNEq	499	105	105	Eq	3	192	497	1392	2087

MNEq: M_0 and M_1 "May Not be Equivalent".
#c: # of lines in c program.
T: Time in milliseconds(ms).
PSE: Number of split-equivalent paths.
#SMT: Number of times SMT solver is called.
SMT_TIME: the time spent on SMT solver.

scheduler. In this experiment, we compare our method with the EVP method [56] to verify the benchmarks listed in Table 4.2. The objectives are (i) to confirm that power of the EVP method is not affected due to new cutpoint selection scheme S_3; (ii) to compare the execution time (in milliseconds (ms)) with the EVP method since our method uses cutpoint selection criteria S_3 while the EVP method uses selection criteria S_1. The result of this experiment is tabulated in Table 4.2. For each benchmark, we record the number of cutpoints (#cutpoint), the number of paths (#path) in both the behaviors (M_0 and M_1) by executing these benchmarks in both the methods (EVP and Our). The 6th and 13th columns, "FEP" (i.e.,#`findEquivalentPath`) represent the number of times `findEquivalentPath`

function is called by the EVP and by our method, respectively. In addition, the maximum length of the formula (in terms of the number of variables along with that of the operations) sent to Z3 SMT solver is tabulated in 8th and 15th columns of Table 4.2 for the EVP and our method, respectively. This formula captures the equivalence formulation of condition of execution of two paths. Our method can establish the equivalence in all these benchmarks. This is reported as 'Eq' in Table 4.2. This result confirms that our cutpoint selection scheme S_3 can also show the equivalence correctly when the control structure has been modified as well as code motions have been applied.

The runtime comparison of our method and the EVP method is shown in 16th and 17th columns for the benchmarks listed in Table 4.2. In general, S_3 simplifies the control structure of the given FSMD hence reduce the number of calls of the function `findEquivalentPath`. The less number of calls for the function `findEquivalentPath` results in less runtime for our method as compared to the EVP method for the benchmarks listed in Table 4.2 except LRU. The benchmarks DIFFEQ and MODN do not have any nested `if-else` block; therefore, both the methods take the same time to show the equivalence. For the benchmarks GCD, TLC, DHRC and BARCODE our method takes less time as compared to the EVP method because our method calls `findEquivalentPath` less or equal number of times. Note that, it is not always true that reduction in the number of the calls for the function `findEquivalentPath` provides the better runtime. For LRU, our method calls `findEquivalentPath` less number of times but takes more time to show the equivalence. As shown in the 8th and 15th columns of Table 4.2, the number of cutpoints reduces from 21 in the EVP method to 11 in our method. As a result, the length of formula at a given state is almost double in our method as compared to the EVP method for LRU. Therefore, our method spends more time in SMT calls for LRU. The experimental result confirms that cutpoint selection criteria S_3 does not increase the runtime exponentially. In fact, the runtime improves in most of the cases.

In our second experiment, all the control dominated benchmarks listed in rows 2–5 of Table 4.3 are taken from [44,104,105]. These benchmarks are used in [44] to show the efficient scheduling of conditional behaviors. The transformed behavior of the benchmark listed in row 1 is obtained by merging the adjacent `if-else` blocks as discussed in Subsection 4.2.3. We obtain the transformed behavior of

the benchmarks listed in rows 2–5 by running Bambu HLS tool [14]. These transformed behaviors represent the scenario where a path in original behavior has been split into more than one path to improve the conditional hardware reuse. For each benchmark, we have reported the number of paths in the path cover, the equivalence decision taken by the EVP method and our method, the run time in milliseconds (ms) of the EVP method and our method, and the number of lines in the C program represents the original behavior. In Table 4.3, we have also reported the number of path split scenario (PSE) has been occurred (column 10), the maximum length of the formula for SMT solver (column 11), #time SMT solver is called (column 12), and the time spent on SMT solver (column 13) by our method. The results of second experiment is tabulated in rows 1—5 of Table 4.3. From these results, it is evident that our proposed method can correctly identify the equivalence even when transformations involve path merging /splitting and merging of states. However, the EVP method fails to prove the equivalence of original and transformed behaviors for these benchmarks, i.e., the EVP method gives false negative results. This is reported as 'MNEq' in Table 4.3. In this case, we do not compare the runtime between the EVP and our method since the EVP method terminates in the middle of its execution by identifying a possible non-equivalence, whereas, our method executes completely and shows the equivalence. The experimental evaluation shows that our method outperforms the EVP method when the paths are merged/split or adjacent conditional blocks having an equivalent conditional expression are merged into one conditional block.

In our third experiment, we take some larger benchmarks from CHStone [55] and Bambu HLS tool [14] to show the scalability of our tool. We obtain a scheduled behavior using Bambu HLS tool for these test cases. We observe that the control structure is modified significantly by Bambu for most of the benchmarks. We have considered a subset of CHStone benchmarks since others contain similar control structure and have similar size for our verification experiment. Specifically, we use the function `BF_cfb64_encrypt` in BLOWFISH, the function `Gsm_LPC_Analysis` in GSM and the function `encrypt` in AES as a source behavior. The results of this experiment is tabulated in rows 6–12 of Table 4.3. The entries related to SMT solver is NULL for MOTION benchmarks since it does not contain any conditional statement. Note that relatively larger benchmark AES takes less time as compared to MIPS because the length of the SMT formula for MIPS is almost 2.5 times of

the same for AES. It may be noted that even for a single path (say β) in one FSMD, we may need to call SMT solver multiple times to check equivalence of condition of execution for each path emanating from the corresponding state of q_{β_s} in other FSMD. In addition, we also use the SMT Solver to find path split/merger scenario and false computation. Therefore, our approach spends on average 43% of total time on checking the equivalence of SMT formulas. It may be noted that the EVP method fails to show equivalence for the benchmarks AES and MIPS since path split/merge scenario arises as shown in column 10 of Table 4.3. Our approach can show the equivalence in less than 6 seconds for all the benchmarks listed in Table 4.3. This experiment shows the scalability of our approach to handle realistic design.

4.8 Conclusions

In this work, a PBEC approach based on value propagation is presented for verification of scheduling of conditional behaviors in HLS. Like the existing value propagation based PBEC approaches [42, 56], our method is capable of handling the control structure modification of input behavior and code motion involving loops. In addition, our method capable of handling the scenario involving path merge/split. Our method can also handle the scenario where adjacent conditional blocks having an equivalent conditional expression are combined into one conditional block. We have also presented a new cutpoint selection scheme to simplify the control structure of the given behavior. The experiments show that our method outperformed the state-of-the-art PBEC approach.

Chapter 5

Improving Performance of a Path-Based Equivalence Checker using Counter-Examples

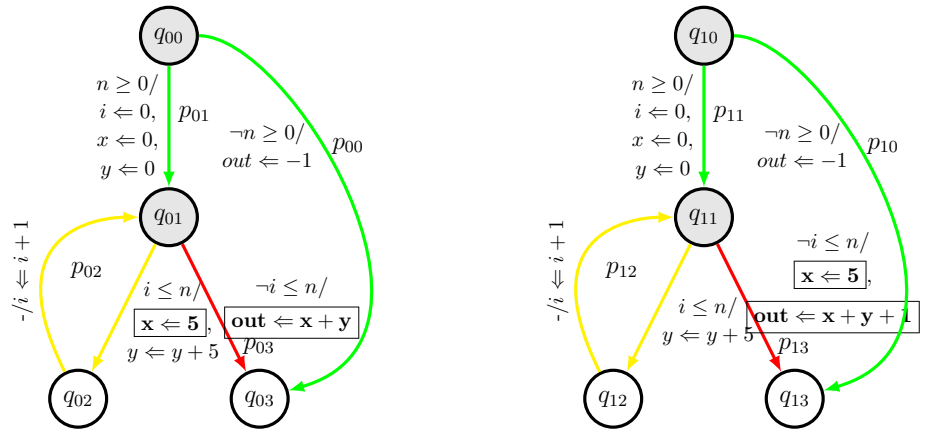
5.1 Introduction

In general, PBEC approaches decompose each FSMD into a finite set of finite paths and the equivalence of FSMDs is established by showing path level equivalence between two FSMDs. In the case of non-equivalence, these approaches do not provide information sufficient for debugging the issue. A counter-example which will demonstrate the non-equivalence between the input behavior to HLS (i.e., source behavior) and the scheduled behavior generated by HLS (i.e., transformed behavior) will add significant value to the adoption of such PBECs. In this case, PBEC approaches can report "Not equivalent" instead of "May Not be equivalent". Equivalence checking of programs is an undecidable problem in general. Therefore it is possible that a PBEC may produce a *false negative* result, i.e., a PBEC may report that two behaviors "May Not be equivalent" but these two behaviors are actually equivalent. The process of generating a counter-example helps to identify some false negative cases of a PBEC approach. Thus, a counter-example generation procedure helps to improve the performance of a PBEC approach.

Specifically, the contributions of this chapter are as follows:

1. We show how the equivalence information of the value propagation based PBEC approach presemted in Chapter 4 can be used to find a *cTrace* in the case of non-equivalence.

2. We show how the Z3 SMT solver [57] and CBMC [58] tool can be used to find a suitable counter-example for a given *cTrace* .



(a) Source behavior M_0 (b) Transformed behavior M_1

Figure 5.1: An example of non-equivalence

3. We show how to improve the performance of the PBEC approach using this counter-example generation framework.

4. An enhanced version of PBEC approach after incorporating our counter-example generation scheme is also presented.

To the best of our knowledge, this is the first work which reports a *cTrace* in the case of non-equivalence and uses it to produce a counter-example and improve the performance of PBECs approaches during verification of the scheduling phase of HLS.

The rest of chapter is organized as follows. A motivating examples highlighting the limitations of a PBEC approach is given in Section 5.2. Section 5.3 focuses on *cTrace* generation. Section 5.4 presents how that *cTrace* can be used to produce a counter-examples. Section 5.5 and 5.6 finally delve into how current PBECs can be enhanced by incorporating our counter-example generation technique. Visualization of *cTraces* generated by our method is explained in Section 5.7. Experimental results are given in Section 5.8. Section 5.9 concludes the chapter.

5.2 Motivations

Consider the input behavior M_0 and its transformed behavior M_1 shown in Fig. 5.1. The operation $x \Leftarrow 5$, a loop invariant for input behavior M_0, is placed after the

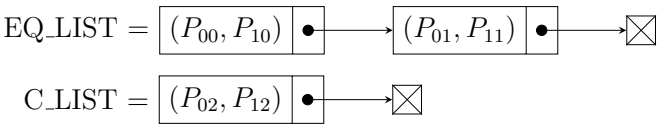
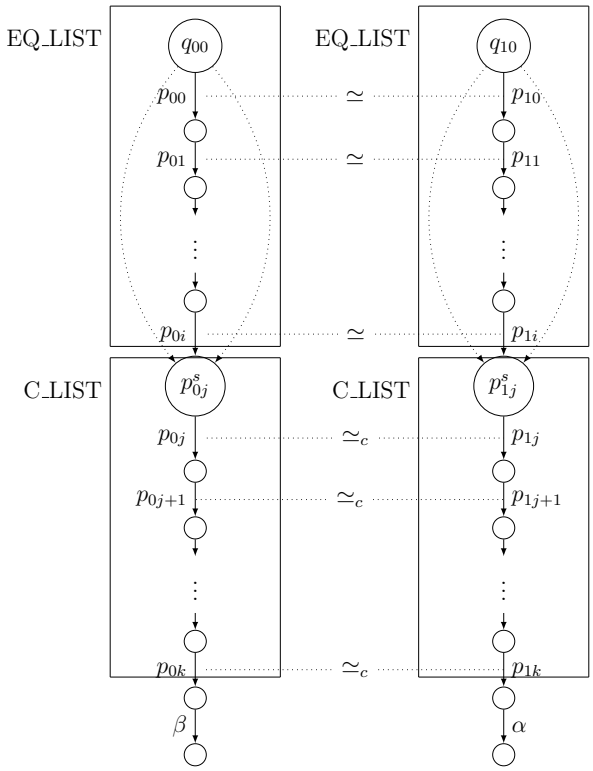


Figure 5.2: List maintained during equivalence checking

loop body in the transformed behavior M_1. Note that the input behavior M_0 and the transformed behavior M_1, shown in Fig. 5.1, are not equivalent since there is mismatch in values of the *out* variable. The PBEC method presented in Chapter 4 reports that behaviors "May Not be equivalent". This method also reports (see Fig. 5.2) that the path pairs (p_{00}, p_{10}) and (p_{01}, p_{11}) are U-equivalent, the path pair (p_{02}, p_{12}) is a candidate for C-equivalence and the path pair (p_{03}, p_{13}) is not equivalent. However, this information is not enough to find the proper reason of non-equivalence. It is desirable the method should produce an input sequence for which both behaviors generate different output values in the case of non-equivalence.

5.3 Counter-Trace Generation

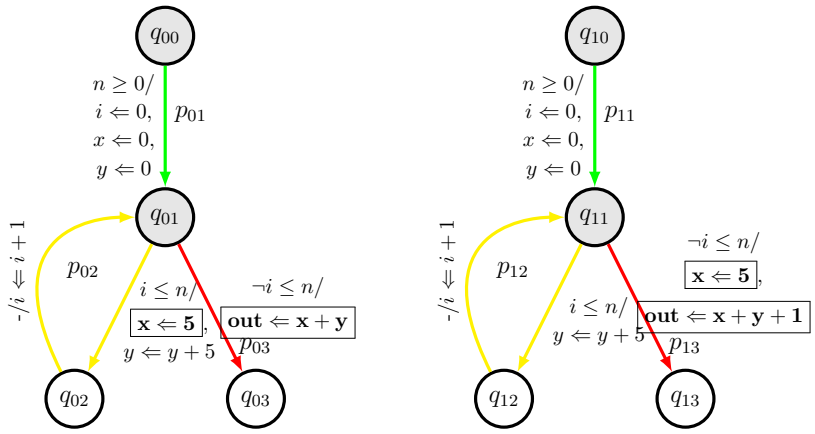
Suppose the source behavior and the transformed behavior are represented as FSMDs M_0 and M_1, respectively. Let us assume that the PBEC approach presented in Chapter 4 fails to find the equivalence of the path β. We now discuss how to generate a unique computation starting from the reset state that leads to the path β. It may be recalled that the PBEC method maintains two lists: EQ_LIST contains equivalent path pairs explored so far and C_LIST contains candidates for conditionally equivalent path pairs. C_LIST is obtained in a DFS manner. So, if we traverse backward from the start state of β, we will obtain a sequence of paths from the set C_LIST. This trace would always be a unique trace. Let the sequence be $\langle p_{0j},\ p_{0j+1},\ ...,\ p_{0k},\ \beta\rangle$ in FSMD M_0 shown in Fig. 5.3(a). The segment of the FSMD M_0 from the reset state q_{00} to the start state of p_{0j}, (say p^s_{0j},) is already proved to be equivalent to its corresponding part in FSMD M_1. However, as shown in Fig. 5.3(a), there might be many paths from q_{00} to p^s_{0j}. For our purpose, we can choose one of the paths from this segment. Let us choose the sequence $\langle p_{00},\ p_{01},\ ...,\ p_{0i}\rangle$ shown in Fig. 5.3(a), where p_{00}



(a) *cTrace* of M_0 (b) *cTrace* of M_1

Figure 5.3: *cTrace* generation using EQ_LIST and C_LIST

starts from the state q_{00} and the path p_{0i} ends at p^s_{0j}. Therefore, the sequence $cTrace = \langle p_{00},\ p_{01},\ ...,\ p_{0i},\ p_{0j},\ p_{0j+1},\ ...,\ p_{0k},\ \alpha\rangle$ is the *cTrace* in the FSMD M_0 shown in Fig. 5.3(a) that we are interested in. From EQ_LIST, we will obtain the paths corresponding to $p_{00},\ p_{01},\ ...,\ p_{0i}$ in FSMD M_1. Let the corresponding paths be $p_{10},\ p_{11},\ ...,\ p_{1i}$, respectively, as shown in Fig. 5.3(b). Similarly, the corresponding paths of $p_{0j},\ p_{0j+1},\ ...,\ p_{0k}$ in the FSMD M_1 can be found using C_LIST. Let the corresponding paths be $p_{1j},\ p_{1j+1},\ ...,\ p_{1k}$, respectively, as shown in Fig. 5.3(b). The *potential* corresponding path of α can also be obtained in the FSMD M_1; let it be α. The PBEC method identifies the potential candidate for equivalence, α, in M_1 in most of the cases. It fails to find α only if there does not exist any path from the corresponding state in M_1 whose condition of execution matches even partially with that of β. In this case, we can take any path from the corresponding state in M_1. Therefore, the corresponding *cTrace* in FSMD M_1 is $\langle p_{10},\ p_{11},\ ...,\ p_{1i},\ p_{1j},\ p_{1j+1},\ ...,\ p_{1k},\ \alpha\rangle$ shown in Fig. 5.3(b).



(a) *cTrace* of M_0 (b) *cTrace* of M_1

Figure 5.4: Counter-trace generation example

Example 14. *Let us consider the input behavior M_0 and its transformed behavior M_1 shown in Fig. 5.4. During the course of equivalence checking the PBEC method presented in Chapter 4 reports that M_0 and M_1, shown in Fig. 5.4 may not be equivalent. The method stores the U-equivalent and candidate for C-equivalent path pairs in the* `EQ_LIST` *and* `C_LIST` *list, respectively shown in Fig 5.2. As explained in Section 5.3, using these lists the generated cTrace of M_0 and M_1 is shown in Fig. 5.4(a) and Fig. 5.4(b), respectively.*

5.4 Counter-Example Generation using Counter-Trace

In this section we explain how a *cTrace* used to produce a counter-examples using Z3 and CBMC tool.

5.4.1 Modeling Counter-trace using Z3 SMT Solver

In the case of non-equivalence reported by the PBEC approach, we generate two *cTraces* as discussed in Subsection 5.3. We then model the equivalence of these *cTraces* as a satisfiability problem. We then apply Z3 to check the satisfiability. If these two *cTraces* are equivalent then Z3 reports `unsat`; otherwise it reports

`sat` and produces a counter-example. The counter-example generation procedure is discussed in detail here.

Let us examine the example of a *cTrace* shown in Fig. 5.4(a). The corresponding *cTrace* in the FSMD M_1 is shown in Fig. 5.4(b). The input to Z3 in SMT-Lib2 language generated for this case is given below.

```
(declare-const n Int)                          ;T0
(declare-const i_0_s Int)                      ;T1
(declare-const x_0_s Int)                      ;T2
(declare-const y_0_s Int)                      ;T3
(declare-const i_1_s Int)                      ;T4
(declare-const x_1_s Int)                      ;T5
(declare-const y_1_s Int)                      ;T6
(declare-const iter_s Int)                     ;T7
(declare-const out_s Int)                      ;T8
(declare-const i_0_t Int)                      ;T9
(declare-const x_0_t Int)                      ;T10
(declare-const y_0_t Int)                      ;T11
(declare-const i_1_t Int)                      ;T12
(declare-const x_1_t Int)                      ;T13
(declare-const y_1_t Int)                      ;T14
(declare-const iter_t Int)                     ;T15
(declare-const out_t Int)                      ;T16
(assert(>= n 0))                               ;T17
(assert(= i_0_s 0))                            ;T18
(assert(= x_0_s 0))                            ;T19
(assert(= y_0_s 0))                            ;T20
(assert(<= i_0_s n))                           ;T21
(assert (> iter_s 0))                          ;T22
(assert(= x_1_s 5))                            ;T23
(assert (= i_1_s (+ i_0_s (* iter_s 1))))      ;T24
(assert (= y_1_s (+ y_0_s (* iter_s 5))))      ;T25
(assert(not(<= i_1_s n)))                      ;T26
(assert(= out_s (+ x_1_s y_1_s)))              ;T27
(assert(= i_0_t 0))                            ;T28
```

Table 5.1: Inverse strength reduction

Init	Op	Incr	Loop	Final
i	$+$	c	n	$i + n * c$
i	$-$	c	n	$i - n * c$
i	$*$	c	n	$i * c^n$
i	$/$	c	n	i/c^n

```
(assert(= x_0_t 0))                                    ;T29
(assert(= y_0_t 0))                                    ;T30
(assert (> iter_t 0))                                  ;T31
(assert(<= i_0_t n))                                   ;T32
(assert(= x_1_t 5))                                    ;T33
(assert (= i_1_t (+ i_0_t (* iter_t 1))))              ;T34
(assert (= y_1_t (+ y_0_t (* iter_t 5))))              ;T35
(assert(not(<= i_1_t n)))                              ;T36
(assert(= out_t (+ (+ x_1_t y_1_t) 1)))                ;T37
(assert (= iter_s iter_t))                             ;T38
(assert (not (= out_s out_t)))                         ;T39
(check-sat)                                            ;T40
(get-model)                                            ;T41
```

We have added statement numbers (having prefix `T`) to aid in our explanation. The variables appearing in the source behavior (Fig. 5.4(a)) are suffixed with `_s`, though the variables appearing in the transformed behavior (Fig. 5.4(b)) are suffixed with `_i`. Since program verification entails checking the equality of output(s) generated by two programs when fed with the same input(s), the variable `n` is not suffixed with either `_s` or `_i`.

The statements `T1`–`T8` and `T9`–`T16` declare the variables appearing in the source and the transformed behaviors, respectively, along with their data type, which is integer for all the variables. The statements `T17`–`T27` and `T28`–`T37` capture the data transformations and the conditions of execution of the paths appearing in the *cTrace* of the two FSMDs. A crucial point to note is that we have considered the loop in the source and in the transformed behaviors to have executed `iter_s` and `iter_i` times, respectively, whose value can be one or more; furthermore, since the conditions of execution and the data transformations of the loops in the two

FSMDs have been found to be equivalent, the two loops must have been executed an identical number of times in the two FSMDs, a fact captured in statement `T38`. Now to generate the counter-example, *we replace the execution of the loop for "iter" times by a single path which captures the equivalent data transformation in each FSMD.* This strategy is adopted because SMT solvers are oblivious to the notion of iterating over loops as done in programs. Moreover, the inputs to an SMT solver must be in single assignment (SA) form (because there is no notion of variable update in SMT languages). Therefore, we have two variables `i_0` and `i_1` in the inputs generated for each FSMD; while the former variable `i_0` is used to store the initial value of variable `i`, the latter variable `i_1` stores the value of `i` after the execution of the loop. Strength reduction [106] is a popular technique for compiler transformation whereby an expression in a loop is replaced by a less expensive operator. Here we are doing reverse of strength reduction to represent a loop in terms of a path in SMT solver. For example the final expression for $i = i + 1$ listed in the loop body of *cTrace* of FSMD M_0 (in Fig. 5.4(a)) will be $i = i + iter * 1$. This final expression is captured in Z3 by the statement `T34`. Intuitively, we apply an *inverse strength reduction* type technique to obtain the final value of a variable given its initial value, increment operator, increment value and loop count as shown in Table 5.1. Accordingly, we get the values for `i_1_s` and `i_1_t` as shown in statements `T24` and `T34`. Similarly, we get the values for `y_1_s` and `y_1_t` as shown in statements `T25` and `T35`.

Finally, we check the equivalence of the output variables `out_s` and `out_i`; note that executing the above code in Z3 reports `sat` for statement `T39` signifying that the values of the variable `out` differs in the source and the transformed behaviors. The statement `get-model` generates the following counter-example for the given pair of *cTrace* shown in Fig. 5.4.

```
sat
(model
  (define-fun iter_t () Int 1)
  (define-fun n () Int 0)
  (define-fun out_t () Int 11)
  (define-fun y_1_t () Int 5)
  (define-fun i_1_t () Int 1)
  (define-fun x_1_t () Int 5)
```

```
  (define-fun y_0_t () Int 0)
  (define-fun x_0_t () Int 0)
  (define-fun i_0_t () Int 0)
  (define-fun out_s () Int 10)
  (define-fun y_1_s () Int 5)
  (define-fun i_1_s () Int 1)
  (define-fun iter_s () Int 1)
  (define-fun x_1_s () Int 5)
  (define-fun y_0_s () Int 0)
  (define-fun x_0_s () Int 0)
  (define-fun i_0_s () Int 0)
)
```

From the above counter-example it is clear that if we initialize the value of n by 0 then the values of *out* differs in the source and the transformed behaviors.

Note that in case of nested loop, if the outer loop is iterated n_1 times and inner loop is iterated n_2 times then a variable with its initial value i, increment operator $+$, increment value c is replaced by a final expression $i + n_1 * n_2 * c$. However, it is always not possible to find a final expression to encode a loop. In general, CBMC is better to model nested loop programs for counter example generation. In the following we discuss the how CBMC can be used to model a counter-trace.

5.4.2 Modeling Counter-trace using CBMC

As discussed in Subsection 5.4.1 for modeling *cTraces* using Z3 SMT Solver an expression in a loop is replaced by a more expensive operator using an inverse strength reduction technique. However, it is not always possible to replace an expression using this technique. Consider the sequence of expressions $x = x + y + i + 5$ and $y = x + y + i + 5$ inside the loop body in the behavior shown in Fig. 5.1(a). In these expressions the value of x and y are dependent on each other hence the strength reduction is not suitable to replace them by less expressive expression. Therefore, in this section we explain that how to model *cTraces* using CBMC instead of Z3. We use CBMC because it symbolically unrolls the loops either completely if possible or to a user-specified depth.

To obtain the counter-example, i.e., assigning suitable value to the inputs, we

rely on CBMC [58]. Specifically, for a given upper bound, CBMC verifies the specified assertions. If any violation of an assertion is detected, a counter-example is generated. Let us consider the *cTraces* as shown in Fig. 5.4(a) and Fig. 5.4(b). The input to the CBMC in C for this case is shown below.

```
1  #include<assert.h>
2  void main()
3  {
4    int i_s,x_s,y_s,n,out_s;
5    int i_t,x_t,y_t,out_t;
6    __CPROVER_assume(n>=0);
7    assert(!(n>=0));
8    // cTrace for M0
9    if(n>=0)
10   {
11     i_s=0;x_s=0;y_s=0;
12     __CPROVER_assume(i_s<=n);
13     assert(!(i_s<=n));
14     while(i_s<=n)
15     {
16       x_s=5;
17       y_s=y_s+5;
18       i_s=i_s+1;
19     }
20     out_s=x_s+y_s;
21   }
22   //cTrace for M1
23   if(n>=0)
24   {
25     i_t=0;x_t=0;y_t=0;
26     __CPROVER_assume(i_t<=n);
27     assert(!(i_t<=n));
28     while(i_t<=n)
29     {
30       y_t=y_t+5;
```

```
31          i_t=i_t+1;
32        }
33        x_t=5;
34        out_t=x_t+y_t+1;
35      }
36      assert(x_s = x_t);// Live Variable
37      assert(y_s = y_t);// Live Variable
38      assert(out_s = out_t);// Output Variable
39  }
```

The variables appearing in the *cTrace* of M_0 (Fig. 5.4(a)) are suffixed with `_s`,whereas the variables appearing in the *cTrace* of M_1 (Fig. 5.4(b)) are suffixed with `_t`. Since program equivalence entails identical output(s) generated by the two programs when fed with the same input(s), the input variable n is not suffixed with either `_s` or `_t`. Lines 3 and 4 declare the variables appearing in the *cTrace* of M_0 and the *cTrace* of M_1, respectively, along with their data type which is integer for all the variables. The lines 8–16 and 18–26 capture the data transformations and the conditions of execution of the paths appearing in the *cTrace* of the M_0 and M_1, respectively. We use `__CPROVER_assume` statements to allow only those computation that satisfy a given condition. For example CBMC first picks the value for n non-deterministically from the domain of integers. The statement `__CPROVER_assume(`$n \geq 0$`)` at line 5 further restricts the range of n for all program computations to be greater than or equal to 0. Note that if there is no computation satisfying the condition, say P, mentioned in `__CPROVER_assume` statement, then all the assertions hold vacuously. We check this by adding `assert(!`P`)` statement after each `__CPROVER_assume` statement so that if one of the `assert(!`P`)` statement is true then we declare that all the possible computations represented by *cTrace* are false computations i.e., they never execute. Finally, we check the equivalence of the live variables (x_s, y_s, x_t, y_t) and output variables (out_s, out_t) using the `assert` statements (lines 27–29).

CBMC is able to automatically determine an upper bound on the number of loop iterations in many cases. It may fail if the number of loop iterations is highly data-dependent. Therefore, to verify the assertions with CBMC we use the following command: `cbmc fileName.c -unwind k --no-unwinding-assertions` where *fileName.c* is the name of the target program, k is the bound on the number of

iterations of the loop in the program called as Unwinding Loop Bound (ULB) and `--no-unwinding-assertions` disables the unwinding assertion check and changes the unwinding assertion to an unwinding assumption. We use the option `--no-unwinding-assertions` so that a counter-example might be found within the small state space generated with the small ULB. If the target program contains a loop then CBMC unwinds the loop k times and check the properties. Note that if there are multiple loops in the program, the bound k applies to all loops. A violation of the property is reported if it is found within k ULB and CBMC will give a counter-example. Otherwise, we iteratively run CBMC with increasing ULBs for the loops until an assertion violation is found or a given time limit is reached.

Note that if the program consists of multiple and possibly nested loops, we simply set the number of loop unwindings globally, that is, for all loops in the program. For every unwinding of an outer loop CBMC unwind each inner loop.

5.5 Incorporation of Results in Equivalence Checking Framework

The PBEC approaches are sound but not complete. Therefore, all the PBECs approaches report that the behaviors "May Not be equivalent" once they fail to prove the equivalence of source and transformed behaviors. Using the output of CBMC, we can actually make the PBEC approach more powerful. In some scenarios, the PBEC approach can report that the behaviors are "Not equivalent" (instead of "May Not") along with a counter-example. Also, in some scenarios, the non-equivalence result reported by the PBEC approach can be proved to be a *false negative* and equivalence checking will proceed further. In the following, we discuss how we can incorporate the CMBC result to improve the equivalence checking framework.

- **Case 1:** *One of the conditions mentioned in* `__CPROVER_assume` *statement is not satisfiable*: In this case, we report to PBEC tool that all the possible computations represented by *cTrace* are false computations. Consequently, we need to proceed further in the equivalence checking process.

- **Case 2:** *The unwinding assertions are valid and CBMC does not find any counter-example*: This means the values of all the live variables and output variables are the same for both *cTraces* . So the non-equivalence reported by the PBEC approach may be a false negative. In this case, we need to proceed further in equivalence checking by declaring the corresponding path pair (α, β) as an equivalent path. This actually helps the PBEC approach to avoid false negative results during the course of equivalence checking.

- **Case 3:** *CBMC reports counter-example for some variables:* This means the data transformation of some variables is not equivalent in the *cTraces* .

  Case 3.1: *A mismatch is found for an output variable*: This is surely a non-equivalence case. So the equivalence checker correctly found the non-equivalence of the behaviors. In this case, the PBEC approach reports that the behaviors are "Not equivalent" along with the counter-example.

  If a mismatch is found only for live variables (which are not output variables), then we cannot conclude definitely that the final outputs of both the behaviors will not be the same. There may be some other operations in the subsequent execution of the FSMDs which will make the behaviors equivalent. Therefore, we need to execute the two programs with the counter-example produced by CBMC and check if the outputs of the two programs are the same or not.

  Case 3.2: *The outputs are the same*: This is not a non-equivalent case. Consequently, we need to proceed further in the equivalence checking process.

  Case 3.3: *The outputs of the two programs are not the same*: This is surely a non-equivalence scenario; in this case, the PBEC approach will report the behaviors are "Not equivalent" along with the counter-example.

- **Case 4:** *CBMC hits the time limit:* In this case, CBMC has failed to generate a counter-example because of time out. So no counter-example will be provided to the user. The PBEC approach reports the behaviors "May Not be equivalent".

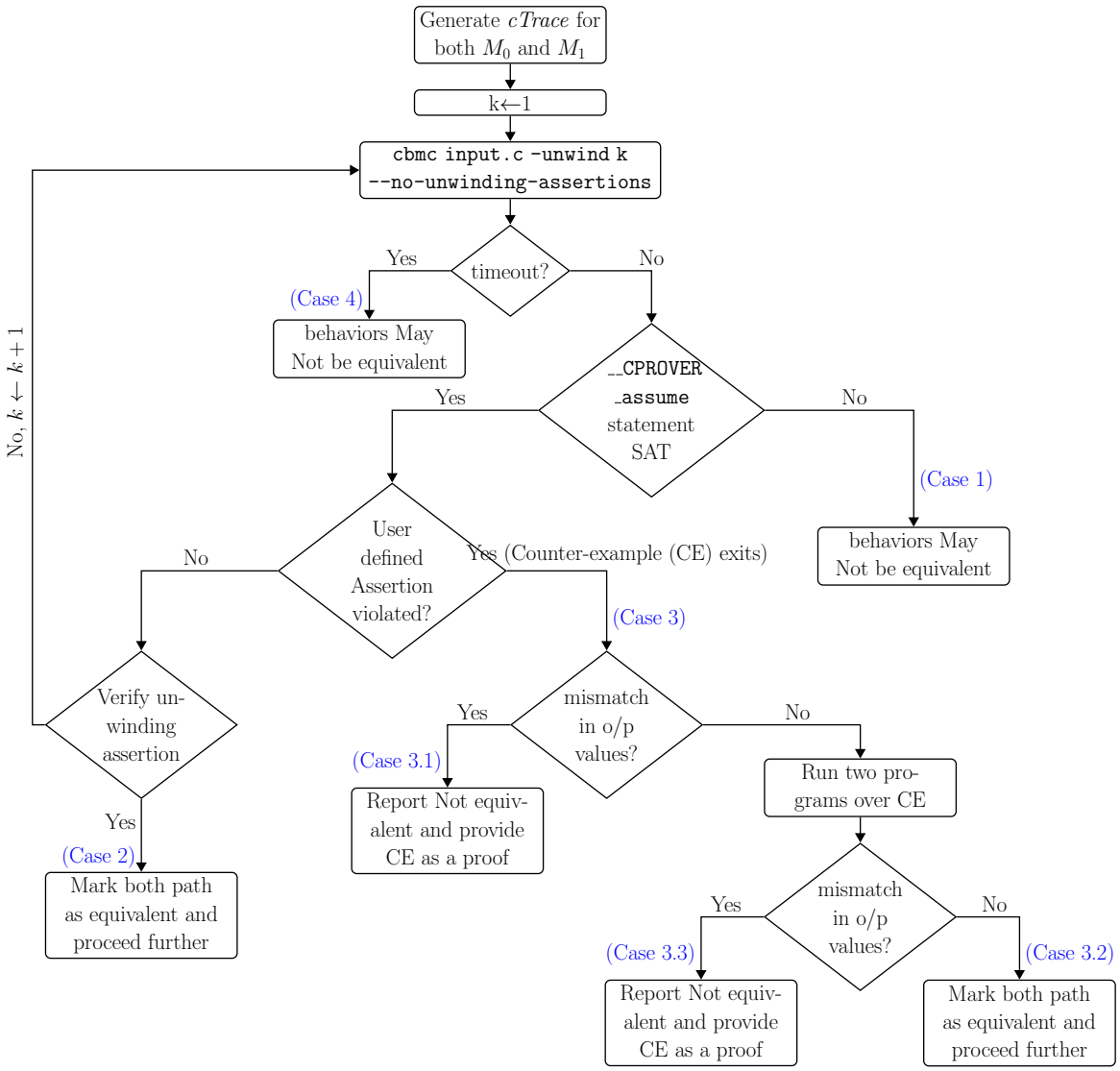


Figure 5.5: Control flow graph of counter-example generation using CBMC and its utilization in a PBEC framework.

5.6 Overall Equivalence Checking Framework

The abstract version of our counter-example generation represented by the function `counterExmapleGenerator` is presented in Algorithm 10. The control flow of Algorithm 10 is given in Fig. 5.5. The function `counterExmapleGenerator` takes as input two FSMDs M_0 and M_1, a path α from the path cover of M_0, a path β from the path cover of M_1, `EQ_LIST` contains equivalent path pairs

Algorithm 10: `counterExmapleGenerator`$(M_0, M_1, \alpha, \beta,$ `EQ_LIST`$,$ `C_LIST`$)$

1 DFS from the start state of α in `C_LIST` to obtain the sequence $\langle p_{0j},\ p_{0j+1},\ ...,\ p_{0k},\ \alpha\rangle$.
2 DFS from the start state of p_{0j} in `EQ_LIST` to obtain the sequence $\langle p_{00},\ p_{01},\ ...,\ p_{0i}\rangle$.
3 Encode the $cTrace = \langle p_{00}, p_{01}, \ldots, p_{0i}, p_{0j}, p_{0j+1}, \ldots, p_{0k}, \alpha\rangle$ and its corresponding *cTrace* in M_1 as C, say "input.c".
4 Initialize the unwinding loop bound (ULB) k to 1.
5 Use `cbmc input.c -unwind k --no-unwinding-assertions` command to invoke CBMC.
6 **if** The condition mentioned in `__CPROVER_assume` is not satisfiable **then**
7 | **return** $\langle$NULL, `False`, `True`$\rangle$; `/* Case 1 */`
8 **else if** All the unwinding assertions along with the user defined assertions are valid **then**
9 | **return** $\langle$NULL, `True`, `False`$\rangle$; `/* Case 2 */`
10 **else if** CBMC produces a counter-example for the assertion belongs to an output variable **then**
11 | **return** $\langle\bar{v}$, `False`, `False`$\rangle$; `/* Case 3.1 */`
12 **else if** CBMC produces a counter-example for the assertion belongs to live variable **then**
13 | Execute both M_0 and M_1 with the values obtained from CBMC as inputs.
14 | **if** outputs are the same **then**
15 | | **return** $\langle$NULL, `False`, `False`$\rangle$; `/* Case 3.2 */`
16 | **else**
17 | | **return** $\langle\bar{v}$, `False`, `False`$\rangle$; `/* Case 3.3 */`
18 | **end if**
19 **else if** CBMC hits the time limit **then**
20 | **return** $\langle$NULL, `False`, `False`$\rangle$; `/* Case 4 */`
21 **else**
22 | Increase ULB by one (i.e., k=k+1) and go to step 5
23 **end if**

and `C_LIST` contains candidates for conditionally equivalent path pairs. The function `counterExmapleGenerator` returns $\langle\bar{v}, Equiv, falseComp\rangle$, where $\bar{v} = \langle v_1, v_2, \ldots, v_n\rangle$ is the input variable list such that v_i represents the value of the input variable v_i, *Equiv* is `True` if $\alpha \simeq \beta$ and `False` otherwise and *falseComp* is `True` if all the computations represented by *cTrace* are false computations and `False` otherwise. In lines 1–2 of Algorithm 10, a *cTrace* is constructed from the `EQ_LIST` and `C_LIST` as discussed in Section 5.3. The *cTrace* is encoded as input to CBMC at line 3. The output generated by CBMC may result in various scenarios

Algorithm 11: `correspondenceChecker`$(M_0, M_1, q_{0i}, q_{1j}, P_0, P_1, W_{csp})$

```
1  foreach path β : (q_0i ⇒ q_0m) in P_0 do
2      if path α : (q_1j ⇒ q_1n) can be found in P_1 such that β ≃ α then
3          W_csp = W_csp ∪ {(q_0m, q_1n)};
4          Insert (β, α) in EQ_LIST.
5      else if path α : (q_1j ⇒ q_1n) can be found in P_1 such that β ≃_c α then
6          if q_0m or q_1n is reset state then
7              return failure;
8          else
9              Insert (β, α) in C_LIST.
10             correspondenceChecker(M_0, M_1, q_0m, q_1n, P_0, P_1, W_csp);
11         end if
12     else
13         ⟨v̄, Equiv, falseComp⟩← counterExmapleGenerator(M_0, M_1, β, α,
             EQ_LIST, C_LIST);
14         if falseComp == True then
15             Proceed Further /* Case 1                                  */
16         else if v̄ ≠ NULL then
17             return Not equivalent; /* Case 3.1                        */
18         else if v̄ == NULL and Equiv == True then
19             Proceed Further /* Case 2                                 */
20         else if v̄ == NULL and Equiv == False then
21             Proceed Further /* Case 3.2                               */
22         else
23             return May Not be Equivalent; /* Case 4                  */
24         end if
25     end if
26 end foreach
27 EQ_LIST = EQ_LIST ∪ {Last member of C_LIST}
28 C_LIST = C_LIST \ {Last member of C_LIST}
29 return success;
```

as discussed in Section 5.5. Lines 6–19 of Algorithm 10 handle these cases.

The enhanced version of `correspondenceChecker` function of the PBEC method presented in Chapter 4 after incorporating the result of the function `counterExmapleGenerator` is presented in Algorithm 11. In case of failure, Algorithm 11 invokes the function `counterExmapleGenerator` (Algorithm 10) at line 13. It may be noted that the PBEC method reports failure under this scenario. If `counterExmapleGenerator` returns a counter-example (i.e., $\bar{v} \neq$ `NULL`) then the

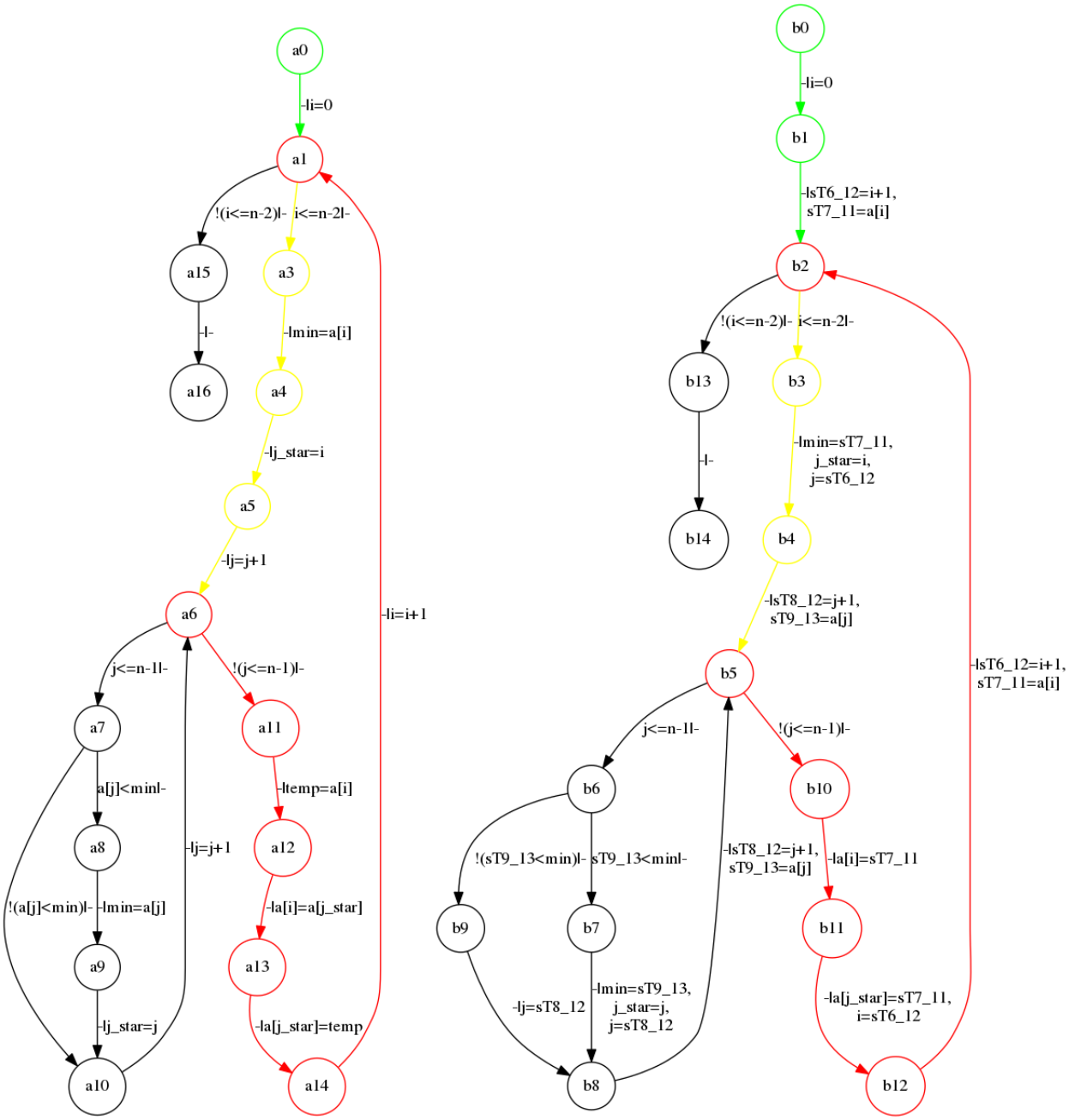
function `correspondenceChecker` returns "Not equivalent" i.e., the two FSMDs are not equivalent (line 17). If CBMC hits the time limit then we cannot decide whether M_0 is equivalent to M_1. Hence the function `correspondenceChecker` returns "May Not be Equivalent" (line 23). If CBMC reports that all the possible computations represented by *cTrace* are false computations (i.e., the variable *falseComp* is `True`) then the function `correspondenceChecker` needs to be modified to handle this scenario (line 15). If CBMC finds the mismatch in the values of a live variable but outputs of the two programs are the same then we do not report the counter-example (line 21). To handle this case also `correspondenceChecker` needs to be modified. If CBMC declares that the path pair (α, β) are equivalent (i.e., the variable *Equiv* is `True`) then it is a false negative result of the `correspondenceChecker` function (line 19). The `correspondenceChecker` function must take some decision to avoid the false negative case in the future.

5.7 Counter-Trace Visualization

Visualization of the *cTrace* can be a great help in case of a mismatch. In this work, we display the trace starting from the reset state till the mismatched path in both the FSMDs using Graphviz [107]. Graphviz is a graph visualization software which can be used to represent graphs and networks as diagrams. For visualization, the internal data structure of an FSMD is stored in a file with *dot* extension – a format which is supported by Graphviz. While visualizing the FSMD using Graphviz, different colors can be used to differentiate between the U-equivalent, candidate C-equivalent and mismatched paths. The following color coding is used to mark a *cTrace* in the FSMD.

1. Green is used to show U-equivalent paths.
2. Yellow is used to color paths which are candidate C-equivalent.
3. Red is used to color a path in the original FSMD for which equivalence is not found and its *most likely* corresponding path in the transformed FSMD (based on the similarity of the conditions of execution).

A *cTrace* typically consists of a green trace, followed by a yellow trace and finally ends in a red trace. The convention is that equivalence of the green trace



(a) Original FSMD (b) Transformed FSMD

Figure 5.6: Two FSMDs before and after scheduling

is found in the other FSMD. So, the correspondence between the green traces is shown in both the FSMDs. The yellow part of the *cTrace* says these are the

candidate C-equivalent paths. This means there is some mismatch of values along this trace. The tool propagates the mismatched value along this yellow trace hoping to identify some compensating transformation which will render the current mismatches into matches in future. Again, candidate C-equivalent paths have a unique correspondence in the other FSMD. So, the correspondence between the yellow traces is shown in both the FSMDs. Then the red path, say β, in the original FSMD M_0 is the path whose equivalent path in the transformed FSMD M_1 is not found. As discussed in Subsection 5.3, the corresponding path, say α, of β can also be obtained. So the correspondence between red paths is also shown in both the FSMDs.

An example is shown in Fig. 5.6 where paths of both the FSMDs are colored with green, yellow and red. In this example, the original FSMD depicts the behavior of selection sort and the transformed FSMD is obtained by running the SPARK tool on the selection sort algorithm. This is an example which reveals a bug in the implementation of copy propagation for array variables in the SPARK tool that was first reported in [71]. As shown in Fig. 5.6, the inner loop $\langle a6 \xrightarrow{j \leq n-1} a6 \rangle$ of the original FSMD and the inner loop $\langle b5 \xrightarrow{j \leq n-1} b5 \rangle$ of the transformed FSMD find the minimum element in the array $A[j \cdots n-1]$ and stored its index in the variable j_star. The inner loop exit path $\langle a6 \xrightarrow{!(j \leq n-1)} a1 \rangle$ of the original FSMD shown in Fig. 5.6(a) swaps the values of $a[i]$ and $a[j\_star]$. But in the inner loop exit path $\langle b5 \xrightarrow{!(j \leq n-1)} b2 \rangle$ SPARK tool fails to swap the values of $a[i]$ and $a[j\_star]$ as shown in Fig. 5.6(b). In addition, the operation $j = i + 1$ is replaced by $j = j + 1$ in the original FSMD, shown in Fig. 5.6(a), by us so that there will be some C-equivalent path in the course of equivalence checking. In Fig. 5.6 the path $\langle a0 \xrightarrow{-} a1 \rangle$ is equivalent to the path $\langle b0 \xrightarrow{-} b2 \rangle$ since the values of all the variables match except the variables that are not common to the two FSMDs, hence colored green. The path $\langle a1 \xrightarrow{i<=n-2} a3 \xrightarrow{-} a4 \xrightarrow{-} a5 \xrightarrow{-} a6 \rangle$ and $\langle b2 \xrightarrow{i<=n-2} b3 \xrightarrow{-} b4 \xrightarrow{-} b5 \rangle$ are yellow as they have a mismatch in the values of the variable j. In Fig. 5.6(a) the path $\langle a6 \xrightarrow{!(j<=n-1)} a11 \xrightarrow{-} a12 \xrightarrow{-} a13 \xrightarrow{-} a14 \xrightarrow{-} a1 \rangle$ does not have an equivalent path in corresponding transformed FSMD, hence it is colored red.

The visualization information can be interpreted as "if you follow the green trace followed by the yellow trace in both the FSMDs, then the equivalent path cannot be found for the red path of the original FSMD in the other FSMD." Moreover, as shown in Section 5.4, we are also generating a counter-example which

Table 5.2: Experimental results with Z3 SMT solver

Benchmarks	Decision		Time (ms)	Lines
	PBEC Method	Our Method		
DIFFEQ	Eq	Eq	184	28
LRU	Eq	Eq	92	247
DCT	MNEq	NEq	107	318
PERFECT	MNEq	NEq	60	63
MODN	MNEq	NEq	96	85
GCD	MNEq	NEq	52	80

will follow the trace shown graphically. With both the information, the user should easily pinpoint the root cause of an error.

5.8 Experimental Results

We have taken the source code of the our PBEC method presented in Chapter 4 and have implemented our counter-example generation procedure on top of it. Once the PBEC method fails to prove the equivalence, a *cTrace* is automatically generated using `EQ_LIST` and `C_LIST` by our method as discussed in Section 5.3. The benchmarks are taken from [42]. The benchmarks are run on a 1.8 GHz Intel i5 processor with 8 GB of RAM with a timeout limit of 60 seconds. We have manually introduced few changes like addition, multiplication or subtraction of a constant to some of the variables in the benchmarks tabulated in rows 3–6 of Tables 5.2 and 5.3 so that source and transformed behaviors become non-equivalent.

In our first experiment, we translate the two corresponding *cTraces* as an input to Z3 SMT solver. The results of our experimentation are tabulated in Table 5.2. For each benchmark, we have reported the equivalence decision taken by the PBEC method presented in Chapter 4 and our method i.e., PBEC with counter-example framework, the number of lines of SMT-Lib2 code generated as input to Z3 SMT solver and the run time in milliseconds (ms) of the PBEC method presented in Chapter 4 and the runtime of our method. For the benchmarks DIFFEQ and LRU, both methods report equivalence which is denoted as 'Eq' in Table 5.2. The objective is to make sure that our implementation does not have any side effect on

Table 5.3: Experimental results with CBMC

Benchmarks	Decision		Time (ms)	Lines
	PBEC Method	Our Method		
DIFFEQ	Eq	Eq	184	28
LRU	Eq	Eq	92	247
DCT	MNEq	NEq	766	185
PERFECT	MNEq	NEq	227	74
MODN	MNEq	NEq	890	137
GCD	MNEq	NEq	100	97
Test Case [108]	MNEq	MNEq	26	32

the existing method. In the benchmarks reported in rows 3–6, our method is able to prove the non-equivalence and denoted as 'NEq'. However, the PBEC method fails to prove the equivalence of source and transformed behaviors. It reports that the behaviors "May Not be equivalent". This is reported as 'MNEq' in Table 5.2.

In our second experiment, we translate the two corresponding *cTraces* as an input to CBMC. The results of our experimentation are tabulated in Table 5.3. In this table column 4 denotes the number of lines in the C program given as an input to CBMC. It is evident form the result that CBMC finds the mismatch in the values of output variable and generates a suitable counter-example with k = 2 loop unwindings. Hence, our method concludes that the behaviors are "Not equivalent".

In our both experiments, we do not compare the runtime between the PBEC method and our method since the PBEC method terminates in by identifying a possible non-equivalence and reports "May not be Equivalent". Whereas, our method uses counter-example generation mechanism to generate a counter-example and reports the "Not equivalent". Both experiments show that with the help of our counter-example generation scheme a PBEC can take strong decisions about the non-equivalence of behaviors.

In our third experiment, we try to explore the false negative scenario of the PBEC method presented in Chapter 4. For this purpose, we have taken the example given in [108] and the result is tabulated in row 7 of Table 5.3. This test case involves the inverse operation [108]. For this test case, the PBEC method

reports that the behaviors "May Not be equivalent". However CBMC does not generate any counter-example and case 2 as discussed in Section 5.6 arises here. CBMC reports that *cTrace* corresponding to these behaviors are equivalent. Our method still reports "May Not be equivalent" since we have not implemented proceed further scenarios. This experiment exposes a false negative case of the PBEC method. It would be an interesting future work to enhance the PBEC method to handle the test cases which involves inverse operations.

5.9 Conclusions

In this chapter, we have presented a counter-example generation mechanism for the PBEC reported in Chapter 4. A similar counter-example generation mechanism can also be developed for other PBEC methods as well. The idea is to reuse the equivalence information of a PBEC method to generate a counter-trace efficiently and then use it to generate a counter-example. We have also shown that a PBEC method can be further strengthened with the counter-example generation mechanism. As shown in the experiments, the PBEC method can take stronger equivalence decision with help of counter-examples. Our counter-example generation mechanism identifies a false negative result of the PBEC method. In the future, we plan to enhance the our method to handle the 'proceed further' (i.e., false negative cases) scenarios identified by our counter-example generation mechanism.

Chapter 6

Security Analysis of Locking during High-level Synthesis

6.1 Introduction

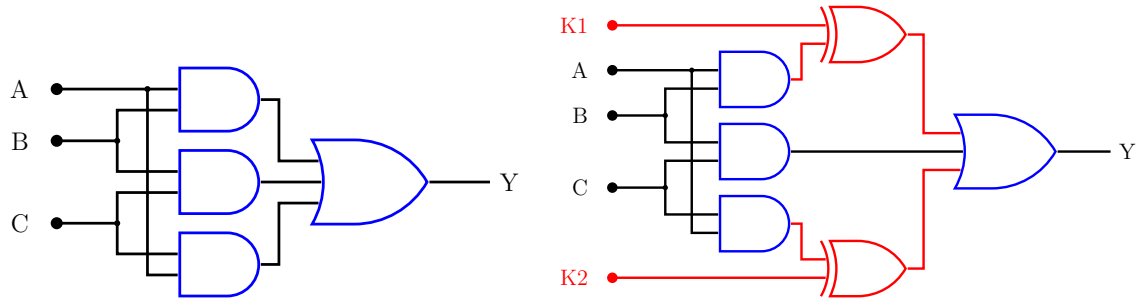
6.1.1 Logic Locking

Many semiconductor companies are *fabless*, i.e., they use offshore third-party foundries to manufacture their chips [29]. While cost effective, the fabless model introduces security concerns. Since the foundry has access to the chip layout, it can reverse engineer the chip's functionality and steal the designer's IP. IP theft of this nature is a serious concern. One approach to preventing IP piracy is *logic locking* [32–34]. In this approach, the circuit functionality is locked using an additional input, called the *key*. Various internal signals of the IC are gated with bits of the key. The IC only functions correctly for a secret key value, known only to the designer, and otherwise produces corrupted outputs. When fabricated chips are received from the foundry (note, the foundry does not know the secret key), the designer activates the chip by loading the correct key in a tamper-proof memory.

Example 15. *Fig. 6.1(a) shows an original netlist of a circuit, and Fig. 6.1(b) shows its functionally locked version through two XOR key-gates. On applying the correct values of the keys (K1=0 and K2=0), the design will produce a correct output; otherwise, it will produce a wrong output.*

6.1.2 Summary of Threats on Logic Locking

Beginning with the SAT attack [46], the past few years have seen a flurry of actions on logic locking, both on the attack and defense side. We note that a provably



(a) Original netlist (b) XOR-based logic locking

Figure 6.1: Logic locking techniques

secure defense against the original SAT attack is still missing. In the SAT attack, the foundry has access to the locked netlist (at the gate-level) and a functioning chip purchased from the market. The attacker then uses the input/output behavior of the functioning chip along with a SAT solver to infer the correct key. First published for breaking combinational circuits, the SAT attack has since been applied to sequential circuits as well [109–111]. However, since the attack operates at the gate-level, these techniques are not scalable to practical designs with hundreds of thousands of gates and flip-flops.

Recent work in [35] has advocated for defenses that perform logic locking during HLS; the resulting RTL locked netlists are *large* and consequently less vulnerable to conventional gate-level SAT attacks. To defeat such RTL locking mechanisms, an attack that works at higher levels of abstraction is desirable. The research question that we attempt to answer is: *"Can one scale the SAT attack to locked RTL?"*

6.1.3 Contributions

We propose an SMT based algorithm to determine the secret key of a locked RTL design obtained through High-level Synthesis [35]. The algorithm models an RTL design as a RTL-FSMD by applying the rewriting approach in [59]. We abstract out the details of the hardware into a behavioral program on which we launch an SMT based attack. Our attack finds distinguishing input patterns iteratively (similar to [46]) to rule out equivalence classes of incorrect keys and

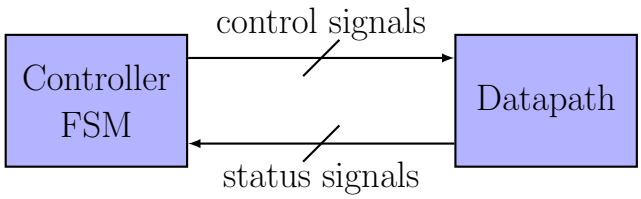


Figure 6.2: RTL structure generated by HLS.

stops when no DIPs are found. For linear arithmetic with m component keys[1], our algorithm is guaranteed to stop within m iterations. Our method works even for non-linear arithmetic since this is supported by the state-of-the-art Z3 SMT solver [57]. Further, our algorithm works on sequential circuits since the analysis is performed on an algorithmic abstraction of the design. We show that the locking keys inserted by TAO [35] can be recovered on HLS benchmarks. To the best of our knowledge, this is the *first* attack on RTL locking.

The chapter is organized as follows. Preliminary concepts, including the attack model, are given in Section 6.2. The TAO approach is discussed in Section 6.3. Our attack/unlocking algorithm is given in Section 6.4. Section 6.5 presents the experimental methodology, results and limitations.

6.2 Backgrounds

This section presents the background required to understand the SMT-based attack.

6.2.1 RTL Structure

The RTL generated by HLS consists of a datapath and a controller finite state machine (FSM) as shown in Fig. 6.2. The datapath consists of registers, memories, functional units (FUs) and their interconnection network. The controller FSM, on the other hand, is a FSM. The RTL operations performed in the datapath are controlled by the controller FSM. In each state, controller assigns 0/1 values to each control signals. As a result a set of RTL operations are performed in the datapath. The datapath sends some status signals (i.e., results of of some

[1]The actual key size is proportional to m.

conditional checks) to the controller. The FSM state transitions depend on those status signals. The RTL is generated by HLS is of this kind of structure.

6.2.2 Attack Model

We assume a malicious foundry that wishes to steal the RTL IP. To protect against this threat, we assume that the designer uses an RTL locking tool like TAO to produce locked RTL[2], performs synthesis and physical design and sends the layout of the locked design to the foundry. As in prior work, we assume the foundry is able to extract the gate-level netlist of the locked chip from its layout. Further, using techniques proposed in [112], we assume the foundry extracts RTL descriptions of the datapath and controller from the gate-level description of the locked chip. Finally, the foundry purchases a functioning (unlocked) copy of the chip from the market and can apply inputs to the chip and observe corresponding outputs (this is the oracle chip). Using this setup, the foundry attempts to recover the secret key to obtain the correct RTL. Note that it is not possible to observe the output of a specific block of a locked RTL design in this setup. Therefore, our attack methodology cannot be used to recover the key of specific block of locked RTL design. Rather, we have to consider the complete locked RTL design in our attack even if a particular block is locked RTL design.

6.3 Motivation

TAO [35] is an algorithm-level locking technique that applies during high-level synthesis. TAO hides selected constants, control branches and datapath operations based on an input locking key K. The key K is provided by the designer through an additional port to the design and partitioned into sub-keys used for each element to lock. The circuit will work correctly only when the correct locking key is given. After applying TAO HLS generates the RTL locked netlist. This locked RTL netlist are large in size and less vulnerable to conventional gate-level SAT attacks. In the following, the TAO locking techniques are briefly presented.

[2]Note that TAO performs locking during high-level synthesis and outputs locked RTL with separate datapath and controller.

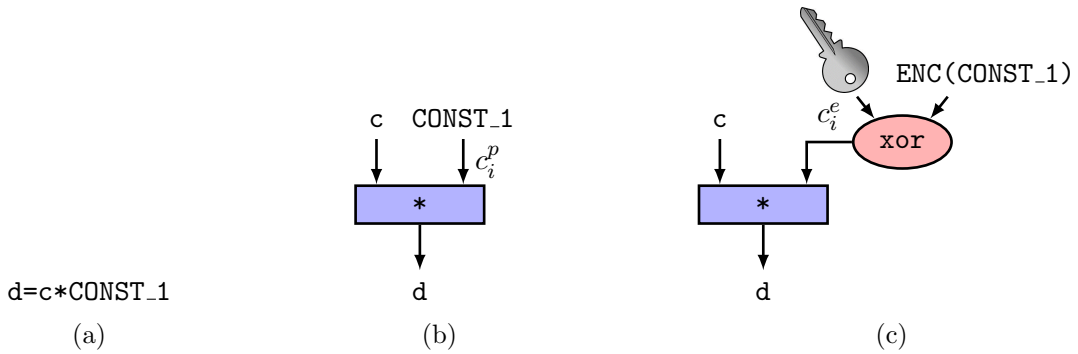


Figure 6.3: An example of constant locking.

6.3.1 Constant Locking

TAO identifies all the constants in the input behavior. It assumes a predefined-number of bits x to implement all constants. Each constant c_i^p of the behavior is locked as $c_i^e = c_i^p \oplus k_i$, where c_i^e is the locked value stored in hardware and k_i is a x-bit key. The correct constant can be obtained by reversing the operation $c_i^p = c_i^e \oplus k_i$.

Example 16. *Consider the constant locking shown in Fig. 6.3. Let say constant* $c_i^p = 5$ *to be stored using 4 bits (0101). This constant can be obfuscated as* $c_i^e = 0110$ *using locking key* $k_i = 0011$*. The correct value is obtained by combing the obfuscated value with input key bits i.e.,* $0101 = 0110 \oplus 0011$*. If a wrong key is provided then the resulting value will be incorrect, but an attacker cannot determine this.*

6.3.2 Branch Locking

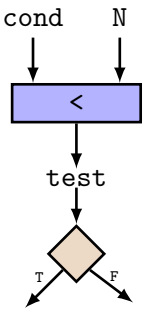
Each branch in the input behavior (and hence in the controller FSM) is locked with a one bit key. If the condition $c_p == 1$ is checked in a control state, the condition is modified as $c_p \oplus k_j == 1$, where k_j is a one bit key. k_j is part of the locking key K and locks this condition checking. The right branch is taken by the controller with the correct k_j.

Consider the if-then statement shown in Fig. 6.4(a). When cond is lesser n, the control transfers to BB2, otherwise it transfers to BB3. Similarly, in Fig. 6.4(b)

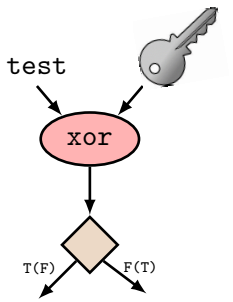
Example 17.

```
if(cond < N){
//go to BB2
 }
else{
//go to BB3
}
```

(a)



(b)



(c)

Figure 6.4: An example of branch locking.

based on the results of the test, control is transferred to BB2 or BB3. In this case an attacker can determine the next block by looking at the result of the test. In Fig. 6.4(c) the control is transferred to the correct block only with correct key bit. For instance, blocks are swapped when key bit is 1. Hence, the attacker cannot determine the actual `true` *(*`false`*) block without knowing the value of the key bit.*

6.3.3 Datapath Locking

TAO adds decoy multiplexer-based interconnections between registers and the functional units. Each MUX is controlled by a key bit k_l. The correct output is connected to 0 or 1 input of this MUX based on the correct value of k_l. This MUX multiplexes the correct and the spurious data flow in each control state. Only with the correct key, the correct operations are performed.

Example 18. *Consider the datapath locking shown in Fig. 6.5. Here the operation $c = a + b$ is the actual operation. In Fig. 6.5c a MUX is added to lock the actual operation and is controlled by key bit. The actual operation is executed if the key value is 0. Otherwise, fake operation $c = a - b$ will be executed. So, the attacker cannot execute the actual operation without knowledge of the key value.*

In our attack implementation, we represent the FSMD model of the RTL design, referred to as an RTL-FSMD model. In RTL-FSMD, V has all the registers and the memories in the design. TAO generated locked RTL is converted into an RTL-FSMD using a *rewriting* (explained in Section 6.4.2). It also embeds the key values and describes how the behavior evolves with different key values.

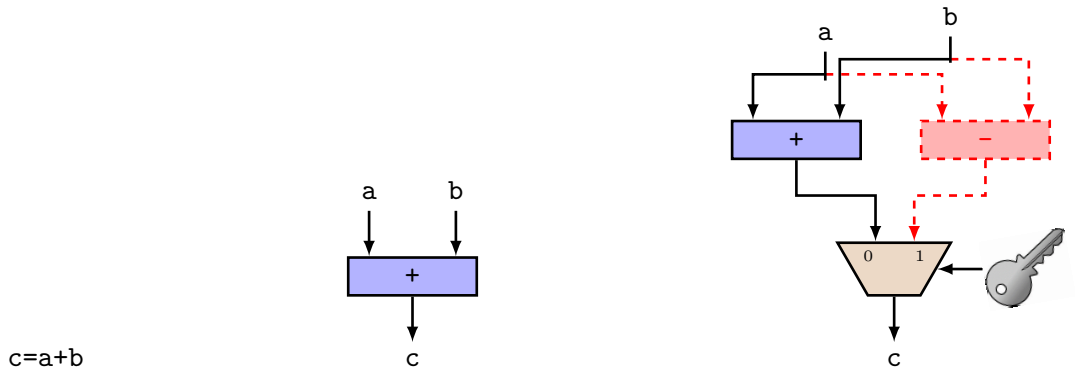
Example 19. *Consider the design in Fig. 6.6. The operations* $r_1 = a + c$ *and* $r_2 = b + d$ *are performed in the datapth in states* $q_1 1$ *and* q_2*, respectively. The MUX in the yellow box is added to lock the first operation and is controlled by the key bit* k_j*. The correct key value is* $k_j = 0$*. Therefore, if* $k_j = 1$ *is supplied,* $r_1 = d + c$ *will be executed producing a wrong result. The locked RTL behavior is shown in the locked FSM with an additional transition between* q_1 *and* q_2*. The key is implicit to the controller FSM. However, when the RTL-FSMD is reverse engineered from the layout, the key* k_l *in unknown and creates additional transitions in the RTL-FSMD.*

6.4 Attack Methodology

6.4.1 Problem Formulation

The objective is to find the locking key K using an SMT solver and by querying an activated IC (the Oracle).

The RTL-FSMD $P(I, O, K) \in \mathbb{Z}^{M+N+K}$ has M primary inputs, N primary outputs and K unknown keys. It represents the input/output relation of the locked RTL design based on the key. $C_O = (I, O)$ is the input/output relation of the activated IC. The attacker can apply inputs to $C_O \in \mathbb{Z}^{M+N}$ and observe the correct output. However, the attacker cannot model the internal behavior of C_O, a *black-box* function $eval(X_i) = Y_i$. For an input X_i, $eval(X_i) = Y_i$ *iff* $C_O(X_i, Y_i)$. While we assume that all inputs/outputs are Integer, this formulation works for



(a) (b) (c)

Figure 6.5: An example of datapath locking.

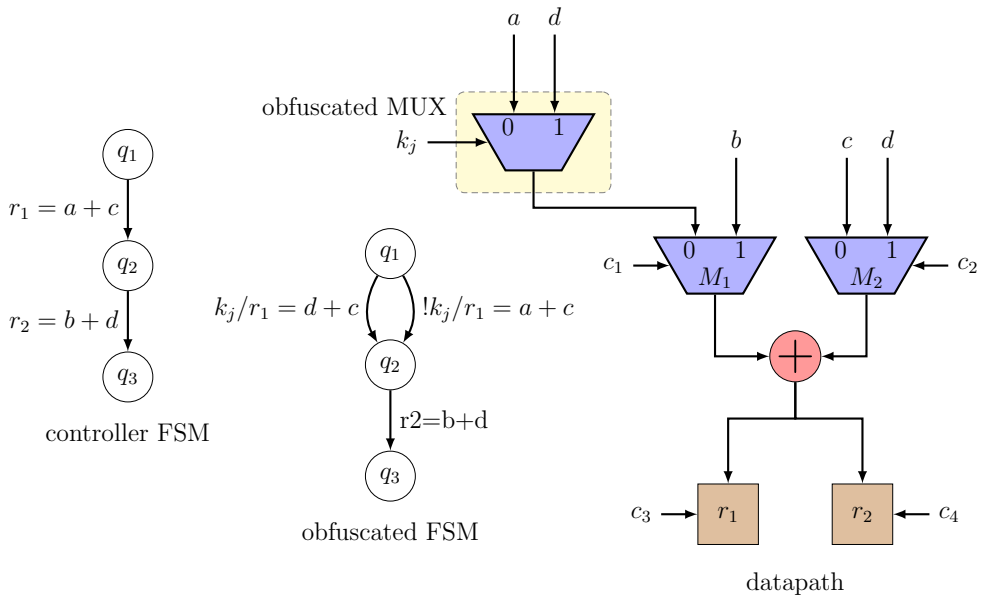


Figure 6.6: An example of TAO obfuscation.

Real numbers.

As shown in Fig. 6.7, the RTL-FSMD consists of a set of states and transitions among the states which represent the control flow. Each transition is associated with a condition and a set of operations that execute in parallel. The data dependencies among the operations represent the data flow. We unroll each loop and the RTL-FSMD is thus a directed acyclic graph. The RTL-FSMD has a *start/reset state* from which any execution starts and terminates. We assume the behavior is deterministic. A *trace* in an RTL-FSMD represents a path from the *reset state* back to the *reset state*. For a trace τ, the *condition of execution* C_τ over $I \cup C \cup K$, where I is the set of inputs, C is set of integer constants and K is the set of unknown keys, represents the symbolic condition that must be satisfied by the initial data state to execute the trace. The C_τ is the weakest precondition of the the trace τ [45]. The *data transformation* D_τ of τ is an ordered tuple of algebraic expressions $\langle e_j \rangle$ over $I \cup C \cup K$ such that e_j represents the value of the output $o_j \in O$ after execution of the trace. C_τ and D_τ can be obtained by the symbolic execution of the trace [38].

The RTL-FSMD consists of a finite set of traces $\{\tau_1, \tau_2, \ldots, \tau_k\}$. The output of an RTL-FSMD will be obtained by the execution of one trace depending on the input values. Each trace has a non-overlapping condition of execution since

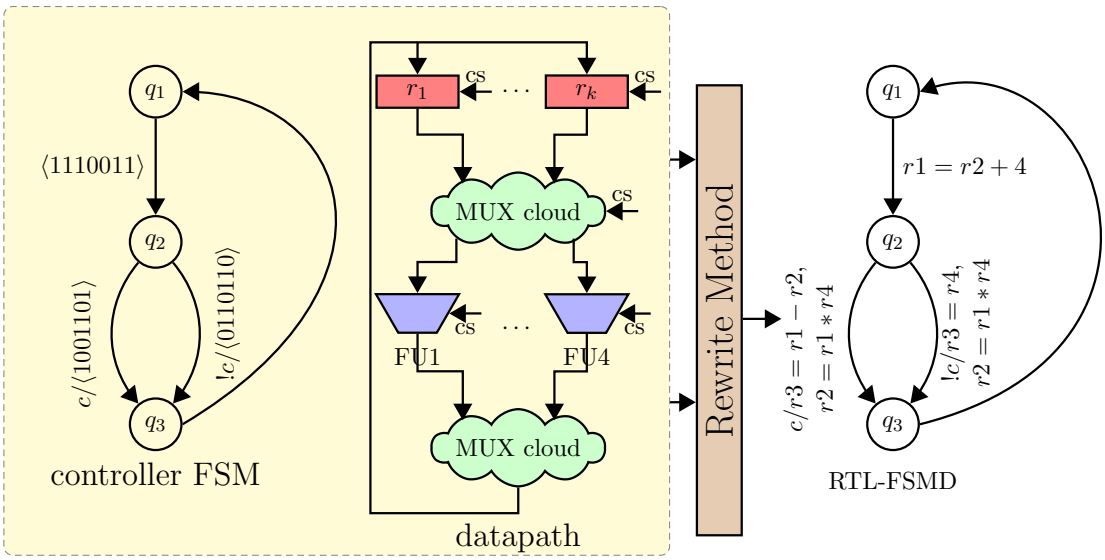


Figure 6.7: RTL-FSMD from RTL using rewriting approach.

the behavior is deterministic. Therefore, the outputs O in the RTL-FSMD can be represented as

$$P(I, O, K): \; O = (ite\ C_{\tau_1} D_{\tau_1}\ (ite\ C_{\tau_2} D_{\tau_2}\ (ite\ \ldots (ite\ C_{\tau_{k-1}} D_{\tau_{k-1}} D_{\tau_k}))\ldots))$$

where $(ite\ C\ D_1\ D_2)$ (aka if-then-else) indicates if the condition C is `True` return the value D_1 else D_2. For a given input I_i and a key K_l and corresponding output O_j, the execution of the P is $P(I_i, O_j, K_l)$. The trace τ_x is executed for this input and key combination, i.e., C_{τ_x} is evaluated to `True` for I_i and K_l. Therefore, $P(I_i, O_j, K_l)$ represents the transformation D_{τ_x} of τ_x, i.e., $P(I_i, O_j, K_l) = D_{\tau_x}$.

6.4.2 Rewriting Method

The HLS-generated RTL consists of a datapath and a controller FSM [53]. In each transition in the FSM, control signals are assigned with value 0/1. Our objective is to identify the corresponding RTL operations performed in the datapath. The control signal assignments in each controller FSM are replaced with corresponding RTL operations. This way, the datapath and the controller details are abstracted out and we have a RTL-level behaviour. The concept is explained in Fig. 6.7. To obtain the RTL operations in each state, we extend the rewriting method presented

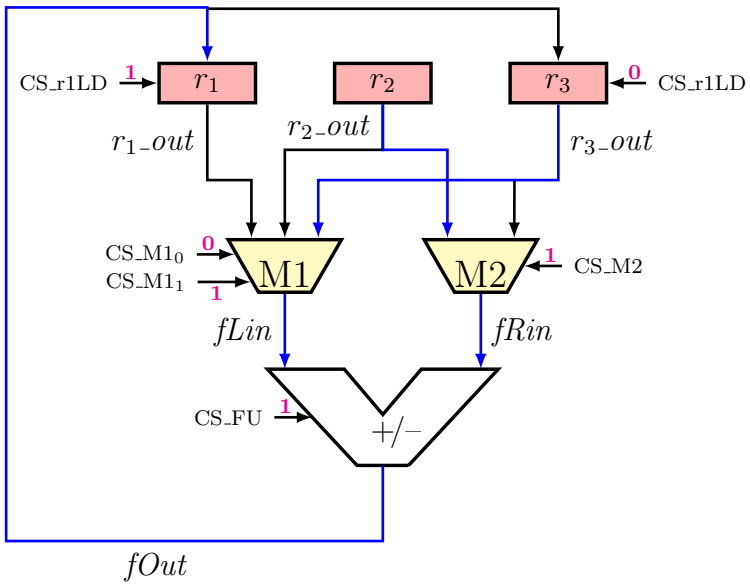


Figure 6.8: Datapath with control signals

in [59] as discussed below.

In the datapath, signal flow is controlled by the control signals. For each datapath module, input → output assignments are termed as micro-operations. For example, for a multiplexer $out = MUX(in_1, in_2, sel)$, there are two possible micro-operations, i.e., $out \leftarrow in_1$ and $out \leftarrow in_2$ and the associated control signal assertions are $sel = 0$ and $sel = 1$, respectively. Given a control signal assignment, we can identify the active micro-operations due this control signal assignment. A micro-operation not associated with any control signal is always active. The *rewriting method* identifies the *spatial sequence* of data flow needed for an RTL operation in a reverse order. The method consists in rewriting terms one after another in an expression. The micro-operations of the form $r \leftarrow r_{in}$ in which a register occurs in the left-hand side (LHS) are found first. Next, the right-hand side (RHS) expression r_{in} is rewritten by looking for an active micro-operation $r_{in} \leftarrow s$ or $r_{in} \leftarrow s_1 < \text{op} > s_2$. Next, s (s_1 or s_2 in the latter case) are rewritten provided they are not registers. The rewriting takes place from left to right in a breadth-first manner and terminates when all signals in the RHS expression are registers.

Example 20. *Consider the datapath shown in Fig. 6.8. In this figure,* r_1*,* r_2 *and*

Algorithm 12: Algorithm to recover the keys.

Input : P, eval
Output: The values of K

```
1 i = 1 ;
2 F_1 = P(I, O_1, K_1) ∧ P(I, O_2, K_2);
3 while sat[F_i ∧ (Y_1 ≠ Y_2)] do
4 |   I_i^d = a DIP value that satisfy [F_i ∧ (Y_1 ≠ Y_2)];
5 |   O_i^d = eval(I^d);
6 |   F_{i+1} = F_i ∧ P(I_i^d, O_i^d, K_1) ∧ P(I_i^d, O_i^d, K_2);
7 |   i = i + 1;
8 end while
9 K = the value of K in the sat assignment of F_i ∧ (Y_1 ≡ Y_2);
```

r_3 are registers, M1 and M2 are multiplexers, and r_1_out, r_2_out, r_3_out, fLin, fRin, and fOut are interconnection wires. The control signal names start with CS. The sequence of rewriting steps for the micro-operation $r_1 = fOut$ is as follows:

$$\begin{aligned} r_1 &= fOut \\ &= fLin - fRin \\ &= r_3\_out - fRin \\ &= r_3 - fRin \\ &= r_3 - r_2\_out \\ &= r_3 - r_2 \end{aligned}$$

6.4.3 Algorithm Description

The problem of finding the distinguishing input pattern can be modelled as follows: *Given two key values K_1 and K_2 and an input I^d, the output obtained is O_1 and O_2, respectively. The input I^d is DIP for K_1 and K_2 iff* $P(I^d, O_1, K_1) \wedge P(I^d, O_2, K_2) \wedge (O_1 \neq O_2)$. Once a DIP is found, the output is obtained from the activated IC. The DIP formulation is strengthened by adding this input/output relation for both K_1 and K_2. This process repeats in an iterative manner until no DIP found. In this time, we will check the SAT of the DIP formula with $(O_1 \equiv O_2)$. Any assignment of K_1 or K_2 for this formula is the correct key. One can recover K using Algorithm 12.

Theorem 7. *Algorithm 12 always terminates.*

Proof. The formula $P(I_i^d, O_i^d, K_1)$ is an equation linking the unknown keys. So,

we add an equation relating the unknown keys in each iteration. Each iteration gets a DIP that rules out one incorrect equivalence classes of keys. Therefore, the equation from each iteration results in an independent equation. If P involves linear arithmetic, one can obtain the values of the K unknown variables by solving the K independent equations connecting them. So, Algorithm 12 finishes in $\|K\|$ steps for linear arithmetic. For non-linear arithmetic, the algorithm resolves when sufficient equations are set up. The search space reduces in each iteration. Therefore, the algorithm completes in a finite number of iterations. The key recovered is consistent with all the observed input/output patterns and thus represents the correct key. ■

6.4.4 Illustrative Examples

In the locked RTL code in Listing 6.1, two constants are locked with k_1 and k_2. Moreover, the condition is locked with a Boolean variable c_1. Assume that $k_1 = 5, k_2 = 3, c_1 =$ `False` in the original program. Our objective is to recover these values from the locked RTL with the help of an oracle.

Listing 6.1: if-else block

```
c = a > b
if(c xor c1)
   out = a + k1
else
   out = b * k2
```

Consider the SMT code in Listing 6.2. The function A in this SMT code models the functionality of the behavior in Listing 6.1. The SMT code to obtain DIP is given by the next three assert statements. *"Does there exist an assignment of a and b such that for two different values of k_1 (i.e., k_{11} and k_{12}) and k_2 (i.e., k_{21} and k_{22}), we have two different outputs?"*. Z3 returns $a = 1, b = 1$ and the corresponding output is 3.

The assertions added in iteration 2 are shown in the first part of the Listing 6.3. The process continues for three more iterations and the assertions added into the DIP model are shown in the rest of Listing 6.3. In the 4th iteration, Z3 returns `UNSAT`. We obtain $k_1 = 5, k_2 = 3, c_1 =$ `False` by reversing `SAT` (i.e.,

$(assert\ (=\ out_1\ out_2)))$ as correct keys.

Listing 6.2: SMT code to obtain the DIP for Listing 6.1

```
(declare-const a Int)
(declare-const b Int)
(declare-const k11 Int)
(declare-const k21 Int)
(declare-const k12 Int)
(declare-const k22 Int)
(declare-const out1 Int)
(declare-const out2 Int)
(declare-const c1 Bool)
(declare-const c2 Bool)
(define-fun G ((a Int) (b Int)) Bool (> a b))
(define-fun A ((a Int) (b Int) (x1 Int) (x2 Int)
               (x3 Bool) (c Bool)) Int (ite
               (xor c x3) (+ a x1) (* b x2)))
(assert (= out1 (A a b k11 k21 c1 (G a b))))
(assert (= out2 (A a b k12 k22 c2 (G a b))))
(assert (not (= out1 out2)))
(check-sat)
(get-model)
```

Listing 6.3: Assertion refinements in successive iterations

```
;Iteration 2:  a = 0, b = 0 → out = 0
;added assertions
(assert (= 0 (A 0 0 k11 k21 c1 (G00))))
(assert (= 0 (A 0 0 k12 k22 c2 (G 0 0))))
Iteration 3:  a = −5, b = −1 → out = −3
;added assertions
(assert (= -3 (A −5 −1 k11 k21 c1 (G −5 −1))))
(assert (= -3 (A −5 −1 k12 k22 c2 (G −5 −1))))
Iteration 4:a = −3, b = −4 → out = 2
;added assertions
```

```
(assert (= 2 (A −3 −4 k11 k21 c1 (G −3 −4))))
(assert (= 2 (A −3 −4 k12 k22 c2 (G −3 −4))))
```

Listing 6.4: Loop

```
s = 0;
for(i = 0;i < 4;i++)
    s = s + h[i] + k;
```

Listing 6.5: SMT code to obtain DIP

```
(declare-const h (Array Int Int))
(declare-const s0 Int)
(declare-const k1 Int)
(declare-const k2 Int)
(declare-const s01 Int)
(declare-const out1 Int)
(declare-const out2 Int)
(define-fun A ((a Int) (b Int) (c Int)) Int
    (+ (+ a b) c))
(assert (= s0 0))
(assert (= out1 (A (A (A (A s0 (select h 0) k1)
               (select h 1) k1) (select h 2) k1)
               (select h 3) k1)))
(assert (= s01 0))
(assert (= out2 (A (A (A (A s01 (select h 0) k2)
               (select h 1) k2) (select h 2) k2)
               (select h 3) k2)))
(assert (not (= out1 out2)))
(check-sat)
(get-model)
Assertion added in Iteration 2:
;h[]=8365, 1796, 8365, 2282 --> out = 20832
(assert (= 20832 (A (A (A (A s0 8365 k1) 1796 k1)
                                8365 k1) 2282 k1)))
(assert (= 20832 (A (A (A (A s0 8365 k2) 1796 k2)
                                8365 k2) 2282 k2)))
```

Loops: Consider the loop in Listing 6.4. In this code, the constant k is locked. In the actual code $k = 6$. Our objective is to obtain the value of k. The SMT code is given in Listing 6.5. Here, the loop is unrolled. After first iteration, Z3 returns the value of $h[] = \{8365, 1796, 8365, 2282\}$, For this input, corresponding assertions are added as shown in the last part of Listing 6.5. In the next iteration, Z3 returns `UNSAT`. The correct value of k is obtained by reversing the SAT problem.

6.4.5 Attack Tool-flow

Fig. 6.9 is our implementation flow. The tool parses the locked RTL generated by TAO using Pyverilog [113] (RTL→ FSMD module). It uses a rewriting method yielding an RTL-FSMD [59] and transforms the RTL-FSMD to feed into the KLEE tool [114] to get the symbolic representation of the outputs as discussed in section 6.4.1. This symbolic representation of the program creates the SAT formulation for DIP. It invokes the SMT tool Z3 [57]. If Z3 cannot prove the SAT/UNSAT of the formula in any iteration, our algorithm fails. If Z3 returns SAT, the corresponding inputs are used to get the correct output using the functional IP. It strengthens the DIP formula with this input/output relation and it calls Z3 again. The algorithm unlocks the keys once Z3 returns UNSAT. The tool flow is automated. RTL → FSMD module is in Python and we write the rest of the tool flow in C++.

We invoke SMT solver Z3 [57] to check for Satisfiability on line 3 of Algorithm 12. SMT solvers require the programs to be in static single assignment (SSA) [115] form. In the SSA form, each variable is assigned exactly once. We model the RTL-FSMD as a formula consisting of the condition of executions and the data transformations of all the traces. This formula represents the one time assignment of each output. So, it is already in the SSA form. This formula is computed using KLEE [114] even if it is symbolic technique that does not require the program to be in SSA form.

6.5 Experimental Results

To evaluate our methodology, we emulated a red team-blue team activity in our experiments. The two teams are in separate institutions. We use three HLS benchmarks - WAKA, ARF and Motion for our experiments. A blue team designer (not

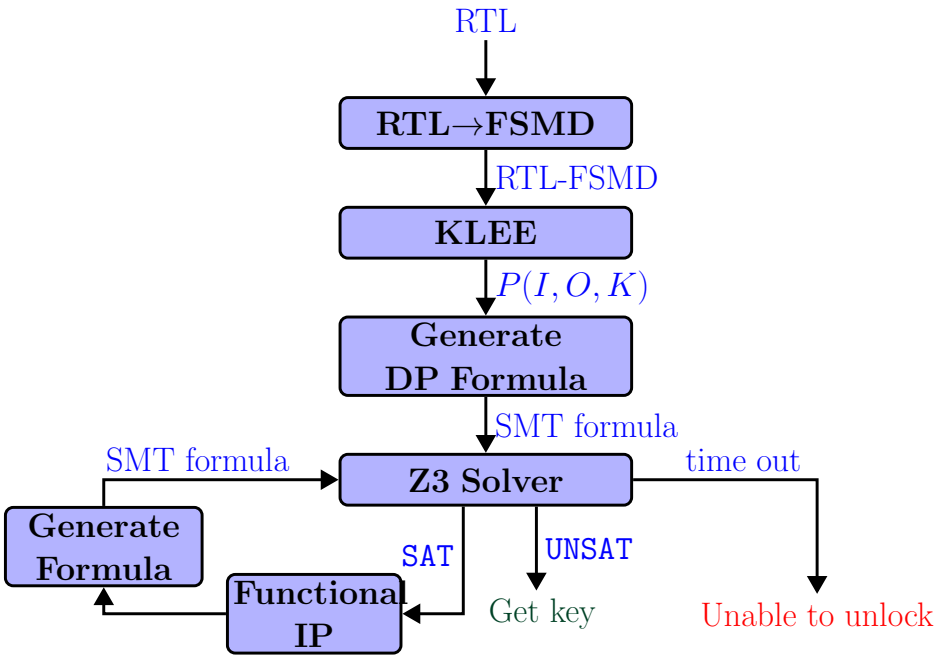


Figure 6.9: Outline of the SMT based unlocking of TAO.

in the red team institution) synthesized these benchmarks with TAO to generate the locked RTL in Verilog [35]. For each test scenario, the number of lines in the Verilog code, the number of multiplications, additions and subtractions in the locked RTL are reported in the columns 2, 3, 4 and 5, respectively, in Table 6.1. As discussed in the section 6.3, TAO applies operation, control and constant locking. The amount of each type of locking is controlled by input parameters. Using these parameters, For each benchmark, the blue team generated several locked designs with differing operations, control-flow statements and constants obfuscated, as shown in columns 6, 7, 8 of Table 6.1, resulting in different key sizes of up to 155 bits, as shown in column 9 of the same table. To check the size of the gate-level circuits targeted by our approach, we synthesized the RTL using Synopsys Design Compiler targeting the SAED 32nm technology. We note that the designs are large with up to 14K combinational cells and 3K sequential cells, as reported in columns 10 and 11, respectively, of Table 6.1.

The red team uses the methodology in this chapter to unlock the designs. The red team unlocking results are tabulated in columns 12-15 of Table 6.1. This includes the number of iterations (Ite) of Algorithm 12 to unlock the key, the

Table 6.1: Results: Unlocking TAO-locked RTL designs.

Bench	LOC	×	+	-	Operations	Conditions	Constants	Key	Comb	Seq	Iterations	Instructions	Time (s)	RAM (MB)
WAKA	753	-	13	7	-	-	3	65	1255	917	4	524	5.16	28
	779	-	23	11	11	4	-	11			5	653	35.46	43
	773	-	23	11	11			9			4	617	92.39	40
	828	-	21	9	9	4	3	73			45	672	1157.13	138
ARF	1431	21	27	10	-	6	-	3	19715	3381	2	6185	517.80	661
	1654	21	27	10	-	-	1	32			2	6863	406.97	576
	1647	21	65	34	65			32			5	6718	>10hrs	-
MOTION	1140	19	11	0	-	-	2	64	13938	2924	5	931	7.01	16
	1239	15	29	10	37	-	-	27			2	885	>10hrs	-
	1250	15	32	10	37	-	4	155			5	924	>10hrs	-

LOC: # of lines in obfuscated Verilog RTL. ×: # of multiplications in Verilog RTL. +: # of adds in Verilog RTL. -: # of subtracts in Verilog RTL. Operations: # of operations obfuscated. Conditions: # of conditions obfuscated. Constants: # of constants obfuscated. Key: # of key bits. Comb: # of combinational cells. Seq: # of sequential cells. Iterations: # of iterations. Instructions: # of instructions executed by KLEE.

number of instructions (Ins) processed by KLEE and the CPU time (Time) and the memory usage (RAM) for each test case. For these experiments, we use Z3 SMT solver version 4.8.5 - 64 bit, with a time out of 10 hours. As shown in Table 6.1, our unlocking algorithm recovers the keys in a few iterations. For successful cases, the *time to unlock is under 30 minutes*. For three cases, Z3 solver times out after a few iterations. We discuss these scenarios in the next section. None of the previously reported combinational unlocking techniques [34,46,50–52] apply in our setting since our locked netlists are sequential. On the other hand, the gate-level SAT attacks on sequential circuits [109–111] reported results for ISCAS'89 and ITC'99 benchmarks, while we scale to much larger benchmarks.

Table 6.2: Results: Unlocking a locked C code.

Bench	Operations	Conditions	Constants	key	Iterations	Instructions	Time (s)	RAM (MB)
WAKA	1	1	5	162	6	306	2888.91	92
	-	-	7	224	8	298	2658.56	120
	2	1	6	195	6	345	3495.51	98
ARF	2	1	1	35	3	1060	1579.77	861
	-	-	4	128	2	1068	400.77	718
	2	1	2	67	3	1142	>10hrs	-
MOTION	2	-	2	66	4	326	11.74	18
	6	-	6	198	8	421	>10hrs	-

Our approach is not limited to HLS-generated designs. It can work on locked C code. For example, in [96], a locked C code is given to a cloud HLS tool to avoid stealing the algorithm IP. To show that we can attack a locked C code, we created several C variants with a large number of key bits and report the results in Table 6.2. For WAKA, we can unlock all cases in one hour. The biggest key that we unlocked is 224 bits. For ARF and Motion, we can unlock up to 128 and 66 bits keys, respectively. For larger key sizes, Z3 times out necessitating scalable approaches.

6.5.1 Discussion of the Results

Handling Time-outs: Solving SMT for arbitrary, non-linear arithmetic over the reals is undecidable [116]. Thus, the SMT solver may not prove the satisfiability of an formula comprising non-linear arithmetic. The SMT solver stops with an *unknown result* or *times out*. Although we did not come across the *unknown* case, we encountered *time outs* for five instances (see Table 6.1 and Table 6.2). We suspect that a time-out implies that no more DIPs exist, i.e., the attack has

terminated although Z3 is unable to prove this formally. To substantiate this, we negate the formula (see step 9 of Algorithm 12) and Z3 returns the *correct* key in all five instances. Thus, even in the few cases that the attack times out, it yields a correct key.

Limitations: We did not implement extraction of arrays from Block RAM in RTL in the RTL → FSMD module yet. Also, functions in the input C code of TAO are in-lined before RTL generation. We will enhance our implementation to support these two features. This will help experiment on larger test cases.

6.6 Conclusions

This work presents an SMT attack to recover the secret key from a locked RTL netlist generated using the TAO RTL locking tool. Compared to gate-level attacks on sequential logic locking, the SMT attack abstracts all hardware details into a behavioral program, scaling to large designs. The behavioral program is an untimed behavior. Therefore, if key is transferred to the registers at some arbitrary time then also our attack can discover the keys. In our attack methodology all the key inputs are initially known. If the key cannot be identified in some scenario, our method is not applicable. The attack is evaluated using a blue team- red team approach, wherein the blue team uses the TAO RTL locking tool to generate locked Verilog RTL along with the executable generated from input C code as an oracle to the red team. The red team unlocked large designs with up to 3K sequential cells and 195 key bits demonstrating the effectiveness of the attack.

Chapter 7

Conclusion and Future Work

Equivalence checking support is critical to the wide adaptation of HLS tools. In this thesis, we designed and developed a path-based equivalence checking framework to verify the correctness of scheduling transformation in HLS. We presented a counter-example generation mechanism to improve debugging the errors in scheduling and to improve the performance of a PBEC approach. We proposed an SMT attack on logic locking during HLS. In this chapter, we conclude the thesis by summarizing our contributions and presenting future work.

7.1 Summary of Contributions

The contributions of this thesis are summarized below:

7.1.1 Translation Validation of Code Motion Transformations Involving Loops during Scheduling

Code motion transformations [7–11] are used in the scheduling phase of HLS tools to improve the quality of synthesis results. Consequently, many research works have been devoted to verifying the semantic equivalence between the original and the scheduled behaviors. Translation validation of behaviors using a path-based equivalence checking method has received attention over the years. The VP method presented in [42] proposed a value propagation based equivalence checking method that can handle the code motion across loop bodies. However, we identified that the VP fails to handles the scenario where some code segment is moved before (after) the loop from inside the loop bodies. We also identified that the state-of-the-art PBEC approaches do not ignore false computation during equivalence checking and produce false negative results. In Chapter 3, we presented an automated formal verification methodology that proves the correctness of HLS

processes involving sophisticated scheduling transformations through value propagation based equivalence checking. The input behavior and the behavior after scheduling have been modeled as FSMDs. The verification problem is treated as the equivalence checking problem of two FSMDs. The method is strong enough to handle the code motion involving loops. The method identifies false computation using the Z3 SMT solver and ignores it during equivalence checking. The algorithm `loopInvariant` is presented which ensures the validity of loop invariant code motion. We also have implemented the method and validated the translations performed by the HLS tool SPARK, and it also uncovered a bug. The experimental outcomes exhibit that the worst-case complexity is not really hit for practical use. In fact, none of the examples hit the worst-case bound.

7.1.2 Verification of Scheduling of Conditional Behaviors in High-level Synthesis

The conditional optimization techniques split a path into multiple paths during scheduling to improve the conditional hardware reuse in HLS. In this case, a path in a behavior is equivalent to the union of the paths in another behavior. In order to handle the path merge/split scenario, a PBEC approach must search the equivalent path in a breadth-first manner. However, the existing PBEC approaches either extend a path or propagate the values in a depth-first manner only to find an equivalent path. Therefore no PBEC approach has been able to deal with path merge/split. In Chapter 4, we presented a PBEC approach for verification of scheduling conditional behavior in HLS. The presented PBEC approach searches for a path in a breadth-first manner as well as a depth-first manner. We introduced path split equivalence, a new notion of equivalence, that is strong enough for verifying the optimization techniques which split a path into multiple paths in the scheduled behavior. A new cutpoint selection scheme is presented which simplifies the control structure of the given behavior. The algorithm `findEquivalentPathAtReset` presented to handle the scenario where conditional merge leads to reset state. The presented method has been proven to be sound but its completeness is being ruled out by the fact that the equivalence of two programs over Integers is inherently undecidable. Experimental results showed that the proposed method could verify designs having complicated control flows. The

scalability of the proposed methods has been shown by running larger benchmark examples. The experimental results also showed that our approach is efficient, and can validate the scheduling transformations on designs in the CHStone benchmarks under 10 seconds.

7.1.3 Improving Performance of a Path-Based Equivalence Checker using Counter-Examples

Many path-based approaches have been proposed for verification of HLS. In the case of non-equivalence these approaches provide only limited feedback to the user. In the case of non-equivalence figuring out the cause of the non-equivalence from the information provided by these approaches is not straightforward and requires human expertise. We presented a counter-example generation mechanism that reports a counter-example in the case of non-equivalence reported by a PBEC approach. The intention is to generate a *cTrace* with the help of the information provided by a PBEC approach and model the *cTrace* to produce a counter-examples using the Z3 or CBMC tool. We also presented a framework to visualize the *cTrace* in the source and the transformed behaviors using the Graphviz tool. This visualization framework helps the user to pinpoint the root cause of an error quickly. We have embedded the counter-example generation mechanism with the PBEC approach presented in Chapter 5. Experimental results also confirm that the counter-example generation mechanism have strengthened the PBEC approach by producing a suitable counter-example in case of non-equivalence.

7.1.4 Security Analysis of Logic Locking during High-level synthesis

In order to overcome the increasing cost of semiconductor fabrication, most of the semiconductor companies are becoming fabless. Fabless IC companies create the project of an IC and outsource the fabrication to a third-party foundry. However, the introduction of third-party manufacturers into the IC supply chain raised concerns over security and trust-related challenges, including overproduction, counterfeiting, IP piracy, and Hardware Trojans. So, the fabless companies are willing to protect the intellectual property of their ICs. Logic locking is a

well-known technique that protects the design against the untrustworthy IC supply chain. Logic locking protects the design by adding some key gates into the original design, so the circuit will not work without a correct key and hides the original design functionality. Most common approaches apply logic locking on the gate-level netlist. However, with the help of this functioning chip, it is often possible to successfully recover the locking key by formulating the attacks as Boolean satisfiability problems (SAT). Pilato et al. proposed a TAO mechanism [35] to lock the IC functionality at a higher level of abstraction. In this approach algorithm-level obfuscation is applied during high-level synthesis and a locked RTL netlist is produced. However, the resiliency of algorithm-level obfuscation has never been investigated. In Chapter 6, we proposed an SMT based algorithm that can determine the key of a locked RTL design inserted by TAO during HLS. The rewriting method has been extended to model an RTL design as an RTL-FSMD. We used KLEE to model the RTL-FSMD as a formula consisting of the condition of executions and the data transformations of all the traces. We execute the SMT attack on the formula generated by KLEE. We implemented the algorithm and the experimental results demonstrated the effectiveness of the attack in unlocking a locked RTL netlist. The results show that our algorithm can unlock large designs with up to 3K sequential cells and 195 key bits within 30 minutes. The SMT attack can also be launched on a locked C code. Finally, we highlighted the challenges and future work required to make SMT attack more practical.

7.2 Future Directions

In this section, possible future works for further improvement of the proposed methods and possibilities of application of our method in other domains are discussed.

- *Enhancement of PBEC framework to handle advanced optimizations:* The loop pipeline, unrolling are the most common optimizations applied by the commercial HLS tools like VIVADO HLS [16], Mentor Graphics Catapult [18] for efficient hardware implementation of image processing applications. Moreover, other loop transformations such as loop merging, loop shifting, and loop vectorization, etc are also applied by the HLS tool. Our current

PBEC approach cannot handle such loop optimizations and give false negative results. A key reason for our inability to handle the loop transformations is that PBEC checks only one iteration of the loop to make decisions. However, it is not sufficient in most of cases to show the equivalence of loop transformations such as loop unrolling, loop merging, etc. We need to loosen the such restrictions without compromising the soundness of the method so that loop transformations can be verified in our PBEC approach.

The arrays are mapped to Random Access Memories (RAMs) by the HLS tool. The numbers of RAMs are limited in the target Field Programmable Gate Array (FPGA) board. Also, their access is restricted. A RAM can be accessed using through the single or dual port during execution. Therefore, multiple small arrays can be merged into one bigger array so that single RAM can be inferred. On the other hand, to accommodate more than one/two reads from an array in a clock cycle, a big array can be split into multiple arrays and then mapped into multiple RAMs. This merging/splitting of arrays can be vertically or horizontally [16]. Our PBEC method cannot handle such merging or splitting of arrays during the scheduling phase of HLS. We would like to enhance our method to handle such optimizations as well.

- *Data-driven approach:* The primary difficulty in the verification of two programs is the loop related transformations. In most of the cases, identifying loop invariant and correspondence of paths in the presence of loop transformations are the key challenges faced by the path-based equivalence checking method. In a recent work [117], the authors used the test cases to guess the loop invariant and also the correspondence of variables of two programs at each loop point. They have also identified likely equivalent paths/corresponding paths between two programs using test cases/data. The formal equivalence of corresponding paths is then proved/disproved using SMT solvers. Another recent work [118] proposes strategies to make the SMT based equivalence checking of two arithmetic expressions scalable. Both of these works are promising since they address the real challenges of program equivalence. We also want to adopt such techniques for our verification of the scheduling problem.

- *End-to-End verification of HLS:* Because of the huge semantic gap between the source behavior in C/C++ and the generated RTL design in Verilog/VHDL, the end-to-end translation validation of HLS is not explored much in the literature. Instead phase-wise verification of HLS, such as scheduling verification [38–42, 66] allocation and binding verification [119] and datapath and controller verification [59], is mostly explored by the researchers. In this thesis also, we have explored the verification of the scheduling phase. However, the primary limitation of the phase-wise verification of HLS is that they need the intermediate synthesis results after each phase from the HLS tool. However, such intermediate synthesis results may not always available or industrial tool may not want to make it public as well. Therefore, an end-to-end translation validation of HLS is the need of the hour for a wide adaptation of HLS tools. We have abstracted a C-like behavior (RTL-C) from the HLS generated RTL in our SMT based attack on HLS obfuscation in Chapter 6. It would be an interesting future work to adapt our PBEC based approach to the equivalence between the input C and the RTL-C for end-to-end verification of HLS.

- *SMT attack on generic RTL designs:* The SMT attack proposed in Chapter 6 unlocks an RTL with an FSMD structure generated by the HLS tool. As discussed in Chapter 6, the HLS generated RTL has a special structure. We have exploited that structure to extracted a C-like high-level behavior from the RTL. However, the abstraction won't work for generic RTL designs. To enhance our attack to handle generic RTL, we can use v2c tool [28] to extract a high-level behavior from the generic RTL and then launch the SMT attack to recover the keys.

- *SMT-resilient obfuscation techniques during HLS:* In general, multipliers create difficulties for any SMT solvers since multiplication operations create instances of non-linear arithmetic. In fact, we found our SMT attack in the Chapter 6 fails to recover the keys for some instances which obfuscates the inputs of a multiplication operation. These results provide some insights on extending the algorithm-level obfuscation techniques to make it SMT attack resistant. Specifically, we aim to identify the hard instances of the SMT solvers and then develop obfuscated RTL by exploiting those instances.

- *Scope of application to other research areas:*
  Evolving Programs: The development of any large scale software system is a gradual process. Validation of such evolving programs is an important problem since any software system moves from one version to another An interesting study would be to check the applicability of the formal methods developed in this thesis to establish the equivalence of evolving programs.

  Automatic Code Generation: Automatic code generation is a standard technique in the area of Software Engineering. Several tools are developed by the research community for generating source code but they do not offer any verification guarantees for the generated code. Testing based approaches for verifying auto-code generators exist [120, 121]. Identifying the scope of applications of the equivalence checking methods developed in this thesis in the verification of code generation process can have a significant impact on the reliability of software.

  Automatic Program Evaluation: Manual assessment of student programs is often slow and inconsistent. Assessment speed can be improved along with consistency by automating the process of evaluation. A survey on automated assessment of programs can be found in [122]. It would be an interesting study to check how our FSMD based equivalence checking method can be enhanced to provide a platform for the automated assessment of programming assignments.

7.3 Conclusion

This dissertation presented a scalable equivalence checking framework for the scheduling transformation in High-level Synthesis. We believe that the proposed framework can greatly contribute to the further improvement of HLS verification. We also introduced an SMT attack to recover the secret keys from a locked RTL netlist generated using the TAO HLS tool. The SMT attack provides some insights how to extend algorithm-level obfuscation techniques to make such attacks difficult.

Bibliography

[1] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-level Synthesis: Introduction to Chip and System Design*. Norwell, MA, USA: Kluwer Academic Publishers, 1992.

[2] G. D. Micheli, *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[3] J. P. Elliott, *Understanding behavioral synthesis: a practical guide to high-level design*. Springer Science & Business Media, 1999.

[4] R. Camposano and W. Wolf, *High-level VLSI synthesis*. Springer Science & Business Media, 2012, vol. 136.

[5] D. C. Ku and G. DeMicheli, *High level synthesis of ASICs under timing and synchronization constraints*. Springer Science & Business Media, 2013, vol. 177.

[6] A. Orailoglu and D. Gajski, "Flow graph representation," in *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, D. Thomas, Ed., Jun 1986, pp. 503–509.

[7] Minjoong Rim, Yaw Fann, and Rajiv Jain, "Global scheduling with code-motions for high-level synthesis applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3, no. 3, pp. 379–392, Sep 1995.

[8] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Dynamically increasing the scope of code motions during the high-level synthesis of digital circuits," *IEE Proceedings: Computer and Digital Technique*, vol. 150, no. 5, pp. 330–337, Sep 2003.

[9] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau, "Conditional speculation and its effects on performance and area for high-level synthesis," in *International Symposium on System Synthesis*, 2001, pp. 171–176.

[10] S. Gupta, N. Savoiu, S. Kim, N. Dutt, R. Gupta, and A. Nicolau, "Speculation techniques for high level synthesis of control intensive designs," in *Proceedings of the 38th Design Automation Conference*, 2001, pp. 269–272.

[11] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau , "Using global code motions to improve the quality of results for high-level synthesis," *IEEE Transactions on CAD of ICS*, vol. 23, no. 2, pp. 302–312, Feb 2004.

[12] S. Gupta, M. Reshadi, N. Savoiu, N. Duff, R. Gupta, and A. Nicolau, "Dynamic common sub-expression elimination during scheduling in high-level synthesis," in *15th International Symposium on System Synthesis*, Oct 2002, pp. 261–266.

[13] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA'11, 2011, pp. 33–36.

[14] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *23rd International Conference on Field programmable Logic and Applications*, Sep 2013, pp. 1–4.

[15] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: A high-level synthesis framework for applying parallelizing compiler transformations," in *16th International Conference on VLSI Design*, Jan 2003, pp. 461–466.

[16] X. Inc. Vivado Design Suite - VivadoHLS. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[17] Cadence. C-to-Silicon Compiler. [Online]. Available: http://www.cadence.com/pr:oducts/sd/siliconcompiler/pages/default.aspx

[18] Mentor Graphics. Catapult C synthesis. [Online]. Available: http://www.mentor.com/products/esl/high_level_synthesis/

[19] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *Tools and Algorithms for the Construction and Analysis of Systems*, 1998, pp. 151–166.

[20] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg, "VOC: A methodology for the translation validation of optimizing compilers," *Journal of Universal Computer Science*, vol. 9, no. 3, pp. 223–247, 2003.

[21] G. C. Necula, "Translation validation for an optimizing compiler," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ser. PLDI'00, 2000, pp. 83–94.

[22] B. Goldberg, L. Zuck, and C. Barrett, "Into the loops: Practical issues in translation validation for optimizing compilers," *Electronic Notes in Theoretical Computer Science*, vol. 132, no. 1, pp. 53–71, 2005.

[23] N. Mansouri and R. Vemuri, "A methodology for automated verification of synthesized RTL designs and its integration with a high-level synthesis tool," in *Formal Methods in Computer-Aided Design*, 1998, pp. 204–221.

[24] R. Radhakrishnan, E. Teica, and R. Vermuri, "An approach to high-level synthesis system validation using formally verified transformations," in *Proceedings of the IEEE International High-Level Validation and Test Workshop*, ser. HLDVT'00, 2000, p. 80.

[25] M. Fujita, "Equivalence checking between behavioral and RTL descriptions with virtual controllers and datapaths," *ACM Transactions on Design Automation of Electronic Systems*, vol. 10, no. 4, pp. 610–626, Oct 2005.

[26] Xiushan Feng and A. J. Hu, "Early cutpoint insertion for high-level software vs. RTL formal combinational equivalence verification," in *2006 43rd ACM/IEEE Design Automation Conference*, Jul 2006, pp. 1063–1068.

[27] A. Leung, D. Bounov, and S. Lerner, "C-to-Verilog translation validation," in *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2015, pp. 42–47.

[28] R. Mukherjee, M. Tautschnig, and D. Kroening, "v2c–a verilog to C translator," in *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2016, pp. 580–586.

[29] S. Heck, S. Kaza, and D. Pinner, "Creating value in the semiconductor industry," accessed May 27, 2019. [Online]. Available: http://www.edn.com/design/integrated-circuit-design/4375454/Is-high-level-synthesis-ready-for-prime-time

[30] J. A. Roy, F. Koushanfar, and I. L. Markov, "EPIC: Ending piracy of integrated circuits," in *2008 Design, Automation and Test in Europe*, Mar 2008, pp. 1069–1074.

[31] M. Rostami, F. Koushanfar, and R. Karri, "A primer on hardware security: Models, methods, and metrics," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1283–1295, Aug 2014.

[32] P. Tuyls, G.-J. Schrijen, B. Škorić, J. van Geloven, N. Verhaegh, and R. Wolters, "Read-proof hardware from protective coatings," in *Cryptographic Hardware and Embedded Systems - CHES 2006*, ser. Lecture Notes in Computer Science, vol. 4249, 2006, pp. 369–383.

[33] J. Rajendran, H. Zhang, C. Zhang, G. S. Rose, Y. Pino, O. Sinanoglu, and R. Karri, "Fault analysis-based logic encryption," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 410–424, Feb 2015.

[34] M. Yasin, B. Mazumdar, J. J. V. Rajendran, and O. Sinanoglu, "SARLock: SAT attack resistant logic locking," in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, May 2016, pp. 236–241.

[35] C. Pilato, F. Regazzoni, R. Karri, and S. Garg, "TAO: Techniques for algorithm-level obfuscation during high-level synthesis," in *IEEE/ACM Design Automation Conference*, Jun 2018, pp. 1–6.

[36] S. Kundu, S. Lerner, and R. K. Gupta, "Translation validation of high-level synthesis," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 29, no. 4, pp. 566–579, Mar 2010.

[37] Y. Kim and N. Mansouri, "Automated formal verification of scheduling with speculative code motions," in *Proceedings of the 18th ACM Great Lakes Symposium on VLSI 2008*, May 2008, pp. 95–100.

[38] C. Karfa, D. Sarkar, C. Mandal, and P. Kumar, "An equivalence-checking method for scheduling verification in high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 3, pp. 556–569, Mar 2008.

[39] C. Karfa, C. A. Mandal, and D. Sarkar, "Formal verification of code motion techniques using data-flow-driven equivalence checking," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 17, no. 3, p. 30, Jul 2012.

[40] C. Lee, C. Shih, J. Huang, and J. Jou, "Equivalence checking of scheduling with speculative code transformations in high-level synthesis," in *Proceedings of the 16th Asia South Pacific Design Automation Conference, ASP-DAC 2011*, Jan 2011, pp. 497–502.

[41] J. Hu, T. Li, and S. Li, "Equivalence checking between SLM and RTL using machine learning techniques," in *International Symposium on Quality Electronic Design, ISQED*, Mar 2016, pp. 129–134.

[42] K. Banerjee, C. Karfa, D. Sarkar, and C. A. Mandal, "Verification of code motion techniques using value propagation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 8, pp. 1180–1193, Aug 2014.

[43] R. Chouksey, C. Karfa, and P. Bhaduri, "Translation validation of loop invariant code optimizations involving false computations," in *VLSI Design and Test*, 2017, pp. 767–778.

[44] O. Peñalba, J. Mendıas, and R. Hermida, "A global approach to improve conditional hardware reuse in high-level synthesis," *Journal of systems architecture*, vol. 47, no. 12, pp. 959–975, 2002.

[45] Z. Manna, *Mathematical Theory of Computation*. Tokyo: McGraw-Hill Kogakusha, 1974.

[46] P. Subramanyan, S. Ray, and S. Malik, "Evaluating the security of logic encryption algorithms," in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, May 2015, pp. 137–143.

[47] X. Xu, B. Shakya, M. M. Tehranipoor, and D. Forte, "Novel bypass attack and bdd-based tradeoff analysis against all known logic locking attacks," in *International conference on cryptographic hardware and embedded systems*, vol. 10529, 2017, pp. 189–210.

[48] M. Yasin, B. Mazumdar, O. Sinanoglu, and J. Rajendran, "Removal attacks on logic locking and camouflaging techniques," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, Aug 2017.

[49] D. Sirone and P. Subramanyan, "Functional analysis attacks on logic locking," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar 2019, pp. 936–939.

[50] Y. Xie and A. Srivastava, "Anti-SAT: Mitigating SAT attack on logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 2, pp. 199–207, Feb 2019.

[51] M. Yasin, A. Sengupta, B. C. Schafer, Y. Makris, O. Sinanoglu, and J. J. Rajendran, "What to lock? functional and parametric locking," in *Proceedings of the on Great Lakes Symposium on VLSI 2017*, ser. GLSVLSI '17, 2017, pp. 351–356.

[52] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. J. Rajendran, and O. Sinanoglu, "Provably-secure logic locking: From theory to practice," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS'17, 2017, pp. 1601–1618.

[53] C. Pilato, S. Garg, K. Wu, R. Karri, and F. Regazzoni, "Securing hardware accelerators: A new challenge for high-level synthesis," *IEEE Embedded Systems Letters*, vol. 10, no. 3, pp. 77–80, Sep. 2018.

[54] R. W. Floyd, "Assigning meanings to programs," *Mathematical aspects of computer science*, vol. 19, no. 1, pp. 19–32, 1967.

[55] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the chstone benchmark program suite for practical C-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

[56] R. Chouksey, C. Karfa, and P. Bhaduri, "Translation validation of code motion transformations involving loops," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 7, pp. 1378–1382, Jul 2019.

[57] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008*, ser. Lecture Notes in Computer Science, vol. 4963, Mar 2008, pp. 337–340.

[58] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2004, pp. 168–176.

[59] C. Karfa, D. Sarkar, and C. Mandal, "Verification of datapath and controller generation phase in high-level synthesis of digital circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 3, pp. 479–492, Mar 2010.

[60] X. Leroy, et al. The CompCert C compiler. [Online]. Available: http://compcert.inria.fr/compcert-C.html

[61] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[62] D. P. Anderson and J. Ainscough, "The verification of scheduling algorithms," in *IEE Colloquium on Structured Methods for Hardware Systems Design*, 1994, pp. 1–7.

[63] N. Narasimhan, E. Teica, R. Radhakrishnan, S. Govindarajan, and R. Vemuri, "Theorem proving guided development of formal assertions in a resource-constrained scheduler for high-level synthesis," *Formal Methods in System Design*, vol. 19, no. 3, pp. 237–273, 2001.

[64] R. Radhakrishnan, E. Teica, and R. Vemuri, "Verification of basic block schedules using RTL transformations," in *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 2001, pp. 173–178.

[65] H. Eveking, H. Hinrichsen, and G. Ritter, "Automatic verification of scheduling results in high-level synthesis," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE'99, 1999, pp. 260–265.

[66] Y. Kim, S. Kopuri, and N. Mansouri, "Automated formal verification of scheduling process using finite state machines with datapath (FSMD)," in *International Symposium on Quality of Electronic Design (ISQED 2004)*, Mar 2004, pp. 110–115.

[67] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers: Principles, techniques and tools," 1986.

[68] E. M. Clarke Jr, O. Grumberg, , and D. Peled, *Model checking*. MIT press, 2002.

[69] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A new symbolic model verifier," in *International conference on computer aided verification*, 1999, pp. 495–499.

[70] J.-B. Tristan and X. Leroy, "Verified validation of lazy code motion," in *Proceedings, PLDI'09*, 2009, pp. 316–326.

[71] S. Kundu, S. Lerner, and R. Gupta, "Validating high-level synthesis," in *Computer Aided Verification*, 2008, pp. 459–472.

[72] T. Li, Y. Guo, W. Liu, and C. Ma, "Efficient translation validation of high-level synthesis," in *International Symposium on Quality Electronic Design, ISQED*, Mar 2013, pp. 516–523.

[73] T. Li, Y. Guo, W. Liu, and M. Tang, "Translation validation of scheduling in high level synthesis," in *Proceedings of the 23rd ACM International Conference on Great Lakes Symposium on VLSI*, ser. GLSVLSI'13, May 2013, pp. 101–106.

[74] R. Camposano, "Path-based scheduling for synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 1, pp. 85–93, Mar 1991.

[75] T. Li, J. Hu, Y. Guo, S. Li, and Q. Tan, "Equivalence checking of scheduling in high-level synthesis," in *Sixteenth International Symposium on Quality Electronic Design*, 2015, pp. 257–262.

[76] Z. Yang, K. Hao, K. Cong, L. Lei, S. Ray, and F. Xie, "Scalable certification framework for behavioral synthesis front-end," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC'14, 2014, pp. 1—-6.

[77] Z. Yang, K. Hao, K. Cong, L. Lei, S. Ray, and F. Xie, "Validating scheduling transformation for behavioral synthesis," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016, pp. 1652–1657.

[78] P. Ashar, S. Bhattacharya, A. Raghunathan, and A. Mukaiyama, "Verification of rtl generated from scheduled behavior in a high-level synthesis flow," in *1998 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (IEEE Cat. No.98CB36287)*, 1998, pp. 517–524.

[79] N. Mansouri and R. Vemuri, "Accounting for various register allocation schemes during post-synthesis verification of RTL designs," in *Design, Automation and Test in Europe Conference and Exhibition*, 1999, pp. 223–230.

[80] J. Dushina, D. Borrione, and A. A. Jerraya, "Formal verification of the allocation step in high level synthesis," in *Forum on Design Languages (FDL'98)*, 1998, pp. 1–10.

[81] C. Karfa, C. Mandal, D. Sarkar, and C. Reade, "Register sharing verification during data-path synthesis," in *2007 International Conference on Computing: Theory and Applications (ICCTA'07)*, 2007, pp. 135–140.

[82] J. A. Roy, F. Koushanfar, and I. L. Markov, "EPIC: Ending piracy of integrated circuits," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE'08, 2008, pp. 1069–1074.

[83] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, "Logic encryption: A fault analysis perspective," in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012, pp. 953–958.

[84] J. Rajendran, H. Zhang, C. Zhang, G. S. Rose, Y. Pino, O. Sinanoglu, and R. Karri, "Fault analysis-based logic encryption," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 410–424, 2015.

[85] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, "Security analysis of logic obfuscation," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC'12, 2012, pp. 83–89.

[86] M. Yasin, J. J. Rajendran, O. Sinanoglu, and R. Karri, "On improving the security of logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 9, pp. 1411–1424, 2016.

[87] S. M. Plaza and I. L. Markov, "Solving the third-shift problem in IC piracy with test-aware logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 6, pp. 961–971, 2015.

[88] A. Baumgarten, A. Tyagi, and J. Zambreno, "Preventing IC piracy using reconfigurable logic barriers," *IEEE Design Test of Computers*, vol. 27, no. 1, pp. 66–75, 2010.

[89] N. Karousos, K. Pexaras, I. G. Karybali, and E. Kalligeros, "Weighted logic locking: a new approach for IC piracy protection," in *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2017, pp. 221–226.

[90] Y. Xie and A. Srivastava, "Mitigating SAT attack on logic locking," in *Cryptographic Hardware and Embedded Systems–CHES 2016*, ser. Lecture Notes in Computer Science, vol. 9813, 2016, pp. 127–146.

[91] M. Li, K. Shamsi, T. Meade, Z. Zhao, B. Yu, Y. Jin, and D. Z. Pan, "Provably secure camouflaging strategy for ic protection," in *Proceedings of the 35th International Conference on Computer-Aided Design*, ser. ICCAD'16, vol. 28, 2016.

[92] M. Li, K. Shamsi, T. Meade, Z. Zhao, B. Yu, Y. Jin, and D. Z. Pan, "Provably secure camouflaging strategy for ic protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 8, pp. 1399–1412, 2019.

[93] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, "Appsat: Approximately deobfuscating integrated circuits," in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2017, pp. 95–100.

[94] Y. Shen and H. Zhou, "Double dip: Re-evaluating security of logic encryption algorithms," in *Proceedings of the on Great Lakes Symposium on VLSI 2017*, ser. GLSVLSI'17, 2017, p. 179–184.

[95] Y. Lao and K. K. Parhi, "Obfuscating dsp circuits via high-level transformations," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 23, no. 5, pp. 819–830, 2014.

[96] H. Badier, J. L. Lann, P. Coussy, and G. Gogniat, "Transient key-based obfuscation for HLS in an untrusted cloud environment," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar 2019, pp. 1118–1123.

[97] K. Z. Azar, H. M. Kamali, H. Homayoun, and A. Sasan, "SMT attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the SAT attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 97–122, 2019.

[98] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[99] K. Wakabayashi and H. Tanaka, "Global scheduling independent of control dependencies based on condition vectors," in *Proceedings 29th ACM/IEEE Design Automation Conference*, Jun 1992, pp. 112–115.

[100] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, Jul 1976.

[101] D. Sarkar and S. C. D. Sarkar, "A theorem prover for verifying iterative programs over integers," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1550–1566, Dec 1989.

[102] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. 1, pp. 121–141, Jan 1979.

[103] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO'04, Mar 2004, pp. 129–142.

[104] H.-P. Juan, V. Chaiyakul, and D. D. Gajski, "Condition graphs for high-quality behavioral synthesis," in *Proceedings of the 1994 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD'94, 1994, pp. 170–174.

[105] J. Li and R. K. Gupta, "An algorithm to determine mutually exclusive operations in behavioral descriptions," in *Proceedings Design, Automation and Test in Europe*, Feb 1998, pp. 457–463.

[106] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, vol. 26, pp. 345–420, 1994.

[107] "Graphviz – Graph Visualization Software," online; accessed 05-Jun-2015.

[108] K. Banerjee, R. Chouksey, C. Karfa, and P. K. Kalita, "Poster: Automatic detection of inverse operations while avoiding loop unrolling," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, May 2018, pp. 175–176.

[109] M. E. Massad, S. Garg, and M. Tripunitara, "Reverse engineering camouflaged sequential circuits without scan access," in *Proceedings of the 36th International Conference on Computer-Aided Design*, ser. ICCAD'17, Nov 2017, pp. 33–40.

[110] T. Meade, Z. Zhao, S. Zhang, D. Pan, and Y. Jin, "Revisit sequential logic obfuscation: Attacks and defenses," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2017, pp. 1–4.

[111] Y. Kasarabada, S. Chen, and R. Vemuri, "On SAT-based attacks on encrypted sequential logic circuits," in *International Symposium on Quality Electronic Design (ISQED*, Mar 2019, pp. 204–211.

[112] J. Rajendran, A. Ali, O. Sinanoglu, and R. Karri, "Belling the CAD: Toward security-centric electronic system design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 11, pp. 1756–1769, Nov 2015.

[113] Shinya, "Pyverilog: A python-based hardware design processing toolkit for verilog HDL," in *Applied Reconfigurable Computing*, ser. Lecture Notes in Computer Science, vol. 9040, 2015, pp. 451–460.

[114] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, vol. 8, 2008, pp. 209–224.

[115] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, Oct 1991.

[116] P. Nuzzo, A. Puggelli, S. A. Seshia, and A. Sangiovanni-Vincentelli, "Calcs: SMT solving for non-linear convex constraints," in *Formal Methods in Computer Aided Design*, Oct 2010, pp. 71–79.

[117] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken, "Data-driven equivalence checking," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA'13, 2013, pp. 391–406.

[118] M. Dahiya and S. Bansal, "Black-box equivalence checking across compiler optimizations," in *Asian Symposium on Programming Languages and Systems*, ser. Lecture Notes in Computer Science, vol. 10695, 2017, pp. 127–147.

[119] C. Karfa, D. Sarkar, C. Mandal, and C. Reade, "Hand-in-hand verification of high-level synthesis," in *Proceedings of the 17th ACM Great Lakes Symposium on VLSI*, ser. GLSVLSI'07, 2007, pp. 429–434.

[120] R. Majumdar and R.-G. Xu, "Directed test generation using symbolic grammars," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 134–143.

[121] P. Sampath, A. Rajeev, S. Ramesh, and K. Shashidhar, "Behaviour directed testing of auto-code generators," in *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*. IEEE, 2008, pp. 191–200.

[122] K. M. Ala-Mutka, "A survey of automated assessment approaches for programming assignments," *Computer science education*, vol. 15, no. 2, pp. 83–102, 2005.

Publications Related to Thesis

1. Ramanuj Chouksey and Chandan Karfa. Verification of scheduling of conditional behaviors in high-level synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–14, 2020.

2. Ramanuj Chouksey, Chandan Karfa, and Purandar Bhaduri. Translation validation of code motion transformations involving loops. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(7):1378–1382, July 2019.

3. Ramanuj Chouksey, Chandan Karfa, Kunal Banerjee, Pankaj Kumar Kalita, and Purandar Bhaduri. Counter-example generation procedure for path-based equivalence checkers. *IET Software*, 13(4):280–285, Aug 2019.

4. Chandan Karfa, Ramanuj Chouksey, Christian Pilato, Siddharth Garg, and Ramesh Karri. Is register transfer level locking secure? In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 550–555, 2020.

5. Ramanuj Chouksey, Chandan Karfa, and Purandar Bhaduri. Improving performance of a path-based equivalence checker using counter-examples. In *International Conference on VLSI Design*, pages 377–382, Jan 2019.

6. Ramanuj Chouksey, Chandan Karfa, and Purandar Bhaduri. Formal verification of optimizing transformations during high-level synthesis. In *Innovations on Software Engineering Conference*, ISEC'19, pages 27:1–27:5, Feb 2019.

7. Ramanuj Chouksey, Chandan Karfa, and Purandar Bhaduri. Translation validation of loop invariant code optimizations involving false computations. In *VLSI Design and Test*, pages 767–778, Dec 2017.