Portland State University

PDXScholar

Dissertations and Theses Dissertations and Theses

Summer 8-5-2015

Scalable Equivalence Checking for Behavioral Synthesis

Zhenkun Yang
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Programming Languages and Compilers Commons

Let us know how access to this document benefits you.

Recommended Citation
Yang, Zhenkun, "Scalable Equivalence Checking for Behavioral Synthesis" (2015). *Dissertations and Theses.* Paper 2461.
https://doi.org/10.15760/etd.2459

This Dissertation is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

Scalable Equivalence Checking for Behavioral Synthesis

by

Zhenkun Yang

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Fei Xie, Chair
Suresh Singh
Feng Liu
Sandip Ray
Fu Li

Portland State University
2015

ABSTRACT

Behavioral synthesis is the process of compiling an Electronic System Level (ESL) design to a register-transfer level (RTL) implementation. ESL specifications define the design functionality at a high level of abstraction (e.g., with C/C++ or SystemC), and thus provide a promising approach to address the exacting demands to develop feature-rich, optimized, and complex hardware systems within aggressive time-to-market schedules. Behavioral synthesis entails application of complex and error-prone transformations during the compilation process. Therefore, the adoption of behavioral synthesis highly depends on our ability to ensure that the synthesized RTL conforms to the ESL description.

This dissertation provides an end-to-end scalable equivalence checking support for behavioral synthesis. The major challenge of this research is to bridge the huge semantic gap between the ESL and RTL descriptions, which makes the direct comparison of designs in ESL and RTL difficult. Moreover, a large number and a wide variety of aggressive transformations from front-end to back-end require an end-to-end scalable checking framework.

A behavioral synthesis flow can be divided into three major phases, including 1) *front-end*: compiler transformations, 2) *scheduling*: assigning each operation a clock cycle and satisfying the user-specified constraints, and 3) *back-end*: local optimizations and RTL generation. In our end-to-end and incremental equivalence checking framework, we check each of the three phases one by one. Firstly, we check

the front-end that consists of a sequence of compiler transformations by decomposing it into a series of checks, one for each transformation applied. We symbolically explore paths in the input and output programs of each transformation, and check whether the input and output programs have the same observable behavior under the same path condition. Secondly, we validate the scheduling transformation by checking the preservation of control and data dependencies, and the preservation of I/O timing in the user-specified scheduling mode. Thirdly, we symbolically simulate the scheduled design and the generated RTL cycle by cycle, and check the equivalence of each mapped variables. We also develop several key optimizations to make our back-end checker scale to real industrial-strength designs. In addition to the equivalence checking framework, we also present an approach to detecting deadlocks introduced by parallelization of RTL blocks that are connected by synthesized interfaces with handshaking protocols.

To demonstrate the efficiency and scalability of our framework, we evaluated it on transformations applied by a behavioral synthesis tool to designs from the C-based CHStone and SystemC-based S2CBench benchmarks. Based on the evaluation results, our front-end checker can efficiently validate more than 75 percent of the total of 1008 compiler transformations applied to designs from the CHStone benchmark, taking an average time of 1.5 seconds per transformation. Our scheduling checker can validate control-data dependencies and I/O timing of all designs from S2CBench benchmark. Our back-end checker can handle designs with more than 32K lines of synthesized RTL from the CHStone benchmark, which demonstrates the scalability of the checker. Furthermore, our checker found several bugs in a commercial tool, underlining both the importance of formal equivalence checking and the effectiveness of our approach.

DEDICATION

To my wife Jialu.

To my parents Xiuzeng and Yunpeng.

In memory of my grandmother Xiuli Zhang (1920 – 2004)

ACKNOWLEDGMENTS

This dissertation could not have been accomplished without generous support from many talented people. I am grateful to all of them from the bottom of my heart.

First and foremost, I would like to express my great appreciation to my advisor, Prof. Fei Xie, for his enlightening guidance and generous support during my Ph.D study. Prof. Xie is an excellent advisor. He taught me how to identify research problems and become an independent researcher. Without his support and encouragement, this dissertation would not have been accomplished. His deep theoretical knowledge and passion about developing practical tools have great influence on my Ph.D research and future career.

I would like to thank Prof. Suresh Singh, Prof. Feng Liu, Dr. Sandip Ray, and Prof. Fu Li for serving on my dissertation committee. Thanks for their advice and sacrifice of valuable summer time for coming to my dissertation defense.

I would like to thank Dr. Kecheng Hao and Dr. Sandip Ray. I benefit a lot from the great infrastructure they built before I joined the group. Many research ideas of this dissertation come from fruitful discussions with them. I sincerely thank them for being excellent collaborators. I am grateful that I have this great opportunity to work with talented group members: Kai Cong, Li Lei, Bin Lin, Disha Puri, Bo Chen, and Christopher Havlicek.

Finally, I would like to thank my parents for their continuous support and endless love. Special thanks to my wife Jialu for her sound and complete love.

Table of Contents

Abstract **i**

Dedication **iii**

Acknowledgments **iv**

List of Tables **vii**

List of Figures **viii**

List of Abbreviations **xi**

Chapter 1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 3
- 1.3 Proposed Equivalence Checking Framework . . . 4
- 1.4 Dissertation Outline . . . 7

Chapter 2 Background **8**
- 2.1 Behavioral Synthesis . . . 8
- 2.2 Symbolic Simulation . . . 9
- 2.3 Formal Equivalence Checking . . . 10

Chapter 3 Front-end Compiler Transformation Checking **12**
- 3.1 Notations and Definitions . . . 12
- 3.2 Equivalence Checking Framework . . . 15
- 3.3 Modular Reasoning across Functions . . . 18
- 3.4 Handling Loops . . . 21
- 3.5 Experimental Results . . . 24

3.6	Related Work	27
3.7	Summary	29
Chapter 4	**Validating Scheduling Transformation**	**32**
4.1	Scheduling Transformation	32
4.2	Formalization	35
4.3	Validation Approach	40
4.3.1	Validating Trace Compatibility	40
4.3.2	Validating I/O Timing	40
4.4	Experimental Results	44
4.5	Related Work	47
4.6	Summary	48
Chapter 5	**Scaling Back-end RTL Generation Checking**	**49**
5.1	Equivalence Checking Framework	49
5.2	Handling Operation Gating Optimization	51
5.3	Handling Global Variables	56
5.4	Experimental Results	58
5.4.1	Performance Evaluation	58
5.4.2	A Behavioral Synthesis Bug	60
5.5	Related Work	61
5.6	Summary	63
Chapter 6	**Interface Synthesis Checking**	**64**
6.1	Interface Synthesis	65
6.2	Deadlock Detection	68
6.2.1	Deadlock Example	68
6.2.2	Deadlock Detection Algorithm	71
6.3	Experimental Results	75
6.4	Related Work	77
6.5	Summary	78
Chapter 7	**Conclusion and Future Work**	**79**
7.1	Conclusion	79
7.2	Future Work	80
References		**82**

List of Tables

Table 3.1 Summary of CHStone Benchmark for Equivalence Checking of Front-end Compiler Transformations 24
Table 3.2 Summary of Evaluation on CHStone Benchmark for Equivalence Checking of Front-end Compiler Transformations 25

Table 4.1 Summary of Evaluation on S2CBench Benchmark 44

Table 5.1 Summary of CHStone Benchmark for Equivalence Checking of Back-end RTL Generation 58
Table 5.2 Summary of Evaluation on CHStone Benchmark for Equivalence Checking of Back-end RTL Generation 59

List of Figures

Figure 1.1 Behavioral synthesis flow 2
Figure 1.2 Behavioral synthesis and verification flow 5

Figure 2.1 A simple function `foo` in C with its symbolic execution tree. (a) Function `foo` in C. (b) Symbolic execution tree of `foo`, where `F` and `X` are symbolic values for `*f` and `x`, `r` denotes the return value, and `nil` denotes that the value is not yet available. 10

Figure 3.1 Framework of checking equivalence between program S and T, which are the input and output of transformation $\mathcal{T}$ respectively. 14
Figure 3.2 A simple function `bar` in C with its symbolic execution tree. (a) Function `bar` in C. (b) Symbolic execution tree of `bar`, where `F` and `X` are symbolic values for `*f` and `x`, `r` denotes the return value, and `nil` denotes that the value is not yet available. 17
Figure 3.3 Symbolic execution tree of function `foo` and `bar`, where `bar` is executed after `foo`, and based on the execution condition of `foo`. 18
Figure 3.4 Global variable usage with sub-function call example. 20
Figure 3.5 A simple function with a loop in C and its IR. (a) Function `f` has an unbounded `for` loop. (b) The IR of `f`, with boxes representing basic blocks, and arrows representing control flow. Control flow merge is implemented via ϕ-instructions in basic block B_2. 21
Figure 3.6 Comparison of success rate on designs of CHStone benchmark without and with cut-loop optimization. The x axis is ordered by the success rate with cut-loop. 26

Figure 4.1 An example of a SystemC thread, which has two superstates in the while loop. 33
Figure 4.2 Cycle-fixed and superstate-fixed scheduling mode for the while loop in thread in Fig. 4.1. 34
Figure 4.3 Extract *superstates* of a thread in SystemC. (a) my_thread is a thread of module dut, wait statements are the boundary of superstates in SystemC. (b) Superstates and their transitions of my_thread. 41
Figure 4.4 An example of incorrect scheduling of signal I/O. (a). Design before scheduling, where a signal output Out is written twice with different values, however, only the last write is visible and valid. (b). The design after scheduling, where two writes of Out are scheduled to two different cycles. For simplicity, we use function add_state() to represent the scheduling transformation will add a new state on that line. 45
Figure 4.5 An example of incorrect scheduling of signal I/O. (a). Design before scheduling, where a local signal sig is written and then read at the same cycle. The read statement takes the old value. (b). The design after scheduling, where the write and read statements are scheduled to two different cycles, then the read takes the new value. For simplicity, we use function add_state() to represent the scheduling transformation will add a new state on that line. 46

Figure 5.1 Dual-rail cycle-based symbolic simulation of a CCDFG and RTL circuit. 49
Figure 5.2 Operation gating example. (a) C code. (b) Data flow graph. (c) Schematic of generated RTL 52
Figure 5.3 Simplified C source code of the MOTION example. 61
Figure 5.4 Bug found in the MOTION example, where an important register is eliminated. (a). Wrong RTL (b). Correct RTL 62

Figure 6.1 Example of interface synthesis. (a). A simple C function with one input and two outputs. (b). Synthesized block diagram of the C function. Input `IN` has an associated valid signal `IN_vld` to indicate when `IN` is ready to be read. Output `O1` has an associated acknowledge signal `O1_ack` to allow the downstream block to acknowledge block 'func' that the output data `O1` has been read. Output `O_2` has both valid and acknowledge signals. 65
Figure 6.2 Timing diagram of the synthesized block 'func'. 66
Figure 6.3 Block diagram of a FIFO with read and write ports. Signals $\overline{\texttt{empty}}$ and $\overline{\texttt{full}}$ indicate the emptiness and fullness of the FIFO. 67
Figure 6.4 Timing diagram of a FIFO with depth of 2. 67
Figure 6.5 A design in C with three blocks. 69
Figure 6.6 Deadlock example: a synthesized design with three blocks. Interface `C` is a FIFO of depth of 2. Interfaces `B` and `D` are FIFOs of depth of 1. (a). FIFOs are initialize to be empty. (b). Status of FIFOs when the design deadlocks. 70
Figure 6.7 Dependency graph of Fig. 6.6. 72
Figure 6.8 Dependency graph example 73
Figure 6.9 Block diagram of `DCT` example. 76
Figure 6.10 Block diagram of `YUV Filter` example. 76

LIST OF ABBREVIATIONS

BDD	Binary Decision Diagram
CCDFG	Clocked Control/Data Flow Graph
CDFG	Control/Data Flow Graph
DFG	Data Flow Graph
EDA	Electronic Design Automation
ESL	Electronic System Level
FIFO	First In, First Out
FSMD	Finite State Machine with Datapath
HLS	High-Level Synthesis
IP	Intellectual Property
IR	Intermediate Representation
PSL	Property Specification Language
RTL	Register Transfer Level
SAT	Boolean satisfiability problem
SEC	Sequential Equivalence Checking
SMT	Satisfiability Modulo Theories
SSA	Single Static Assignment
STP	State Transition Partition
SVA	SystemVerilog Assertions

Chapter 1

INTRODUCTION

1.1 MOTIVATION

With the ever-growing complexity in modern VLSI systems, designing high-quality hardware at Register-Transfer Level (RTL) under high time-to-market pressure is challenging. Behavioral synthesis, also called high-level synthesis (HLS), is the process of compiling an Electronic System Level (ESL) design to an RTL implementation. ESL specifications define the design functionality at a high level of abstraction (*e.g.*, with C/C++ or SystemC), and thus provide a promising approach to address the exacting demands to develop feature-rich, optimized, and complex hardware systems within aggressive time-to-market schedules. Recent years have seen the adoption of behavioral synthesis in industry. A typical example is how Google uses a behavioral synthesis tool to design the G2 VP9 hardware decoder [72] to implement the VP9 video compression standard.

As shown in Fig. 1.1, a behavioral synthesis takes a design specified in high-level languages, applies a sequence of transformations, and generates targeted RTL netlist. A typical behavioral synthesis flow can be roughly divided into three phases: *front-end*, *scheduling* and *back-end*. The front-end primarily entails compiler transformations; the goal is to reduce code complexity of the generated design, maximize data locality, *etc.* [19], and transform the design into a form more suitable for resource allocation and control synthesis. The scheduling transformation

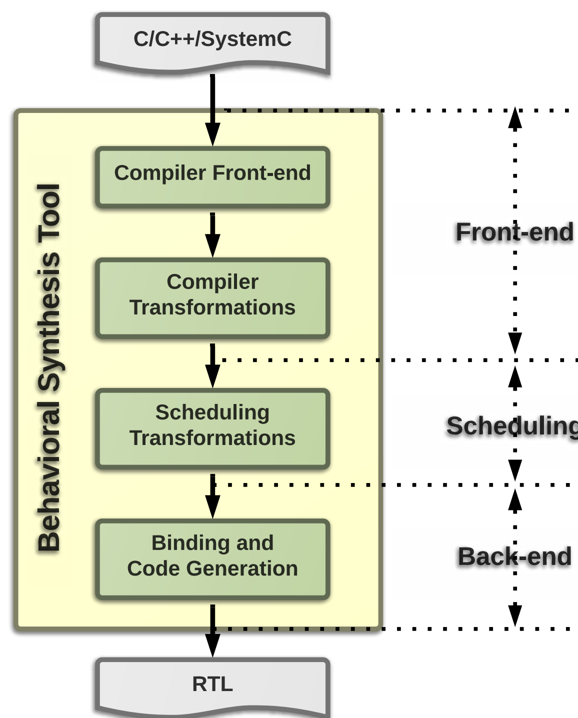


Figure 1.1: Behavioral synthesis flow

schedules each operation a clock cycle to execute. The scheduled design needs to satisfy user-specified timing and resource constraints at the same time. The back-end entails local, sometimes manual, optimizations for a number of metrics, *e.g.*, performance and power consumption, and generates RTL at the end.

Behavioral synthesis provides two major advantages over traditional design methodology: accelerated development and fast verification. Firstly, development of the functionality at a higher abstraction level is much more efficient than development at RTL. Secondly, functional verification can be done at high-level, which is usually hundreds of times faster than verification at RTL in terms of runtime. However, the expensive verification efforts of generated RTL can be saved only if we guarantee that behavioral synthesis tool generates correct RTL from the ESL specification. Therefore despite that there are several commercial behavioral synthesis tools available, the adoption of behavioral synthesis critically depends on

our ability to ensure that the synthesized designs indeed correctly implement the ESL specifications. The goal of this research is to provide a formal equivalence checking support for behavioral synthesis, with the emphasis on scalability and end-to-end checking ability.

1.2 PROBLEM STATEMENT

Real industrial designs are typically thousands or even tens of thousands of lines of `C/C++` code. The performance (runtime and memory usage) of the equivalence checker should scale to large industrial designs. To provide a scalable end-to-end formal equivalence checking support for behavioral synthesis, we need to decompose the verification flow into phases that are defined in the behavioral synthesis flow, and address the following challenges for different phases.

- How to check front-end compiler transformations? Front-end transformations constitute the majority of synthesis transformations. It is typical that more than a hundred transformations are applied to a design. Furthermore, they are applied aggressively under delicate and implicit invariants, and tend to be complex and error-prone.

- How to validate scheduling transformation? Scheduling transformation is usually a very aggressive, complicated, thus error-prone transformation to meet the timing and resource constraints. It entails sophisticated heuristics to minimize register usage, improve timing and possibilities for sharing. The scheduling transformation not only needs to preserve control and data dependencies of the original design, but also needs to accommodate user specified timing constraints: either compiler directives or `wait` statements explicitly specified in the design. In this dissertation, we don't consider pipelining in the scheduling transformation.

- How to make the back-end checking framework scale to industrial-strength designs? Previous work [36, 64] proposed a dual-rail symbolic simulation framework to check the equivalence between the design after scheduling transformation and the synthesized RTL. However, the framework is inadequate to handle design and implementation optimizations, therefore it is not scalable to real industrial-strength designs. In order to improve the scalability of the framework, we need a robust approach to handling the design and implementation optimizations.

- How to detect deadlocks introduced by parallelization of RTL blocks that are connected by synthesized interfaces with handshaking protocols? Behavioral synthesis tools allow designers to synthesize concurrent RTL blocks from a sequential high-level specification, and to synthesize interfaces between blocks into pre-defined interface components with handshaking protocols. Allowing blocks to run concurrently may introduce deadlocks. Therefore, we need scalable approach to detecting deadlocks in the synthesized RTL implementations.

1.3 PROPOSED EQUIVALENCE CHECKING FRAMEWORK

Because of the huge semantic gap between the ESL specification and the generated RTL implementation, and the fact that a large number of transformations will be applied to the ESL design, it is extremely hard to check equivalence of the ESL design and the RTL implementation directly. This research focuses on the ability of checking the entire behavioral synthesis flow and the scalability to industrial-strength designs. Instead of checking the ESL and RTL directly, we employ an incremental checking approach, taking advantage of the similarity of the design representations before and after each transformation. We establish intermediate

equivalence points in the checking flow by decomposing the checking into a sequence of checks, one for each transformation applied by the behavioral synthesis tool. This incremental approach is essential to the scalability of our framework.

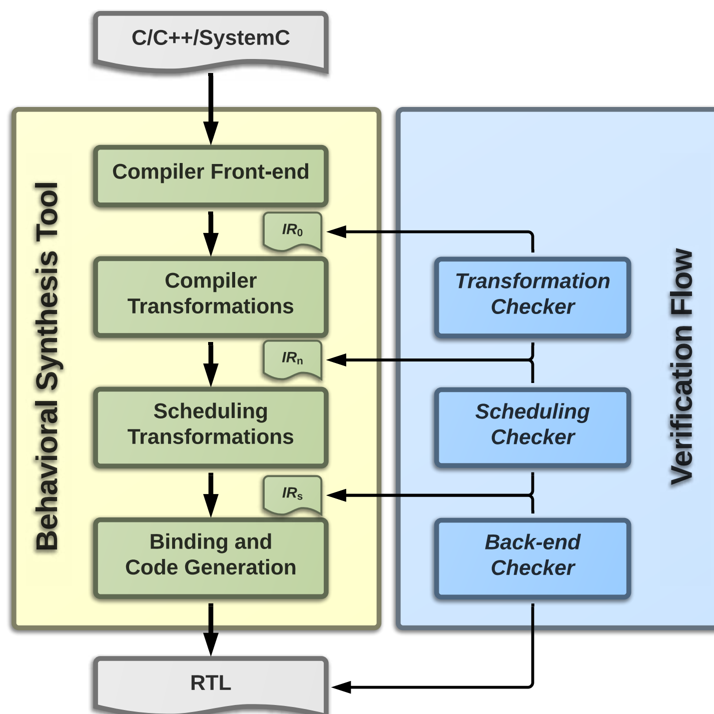


Figure 1.2: Behavioral synthesis and verification flow

As discussed in Section 1.1, a behavioral synthesis tool mainly has three phases: *front-end*, *scheduling* and *back-end*. To build a scalable end-to-end formal equivalence checking framework, we also partition our verification flow shown in Fig. 1.2 into the following three parts.

1. Front-end transformation checker checks the equivalence of the input and output programs of each transformation applied. We use symbolic execution techniques to explore paths of the input and output programs of each transformation. We then check that under the same path condition, whether they have the same observable behavior. Our incremental approach checks the

front-end transformations by decomposing them into a sequence of checks, one for each transformation applied. We identify two major sources to path explosion problem in symbolic execution: *subroutine calls* and *loops*. We propose a compositional checking approach to checking a design that consists of a number of functions on a function-by-function basis, which greatly reduces the checking complexity. We use *cut-loop* optimization to handle path explosions introduced by loops.

2. Scheduling checker validates that 1) every operation in the design must be scheduled to be executed, 2) control and data dependencies of the design are preserved after scheduling, and 3) user specified timing constraints are satisfied. We identify the properties need to be checked for different I/O scheduling modes. and propose different algorithms to check the properties.

3. Back-end checker checks the equivalence of the scheduled design and the generated RTL. We symbolically simulate them cycle by cycle, and compare the mapped variables (including inputs and outputs) at the end of each cycle. Design and implementation optimizations employed either by the designers or by the behavioral synthesis tool complicate the equivalence checking problem. In particular, interface synthesis of global variables of a design and operation gating are hurdles to the scalability of our checker. We use a compositional equivalence checking approach together with automatic inference of the extended signature of each module to handle global variables. We propose a relaxed equivalence checking algorithm to tolerate local and irrelevant in-equivalences introduced by operation gating. These two optimizations are essential to make our checker scale to industrial-strength applications.

In addition to the three-phase equivalence checking framework, our deadlock detection approach takes the design representation after scheduling transformation and the result of interface synthesis, automatically generates assertions to capture

deadlock conditions. The generated assertions can be used in assertion-based verification (simulation or formal verification tools) to detect deadlocks.

1.4 DISSERTATION OUTLINE

The rest of this dissertation is organized as follows: Chapter 2 provides background on *behavioral synthesis*, *symbolic simulation* and *equivalence checking* techniques employed in hardware verification. In Chapter 3, we present our equivalence checking framework for front-end compiler transformations in behavioral synthesis. We also present several optimizations to handle the path explosion problem. In Chapter 4, we present our validation approach to scheduling transformation, including validation of control and data dependencies and I/O interface timing under different scheduling modes. In Chapter 5, we present several optimizations to make our back-end equivalence checker scale to industrial applications. We present an assertion based verification approach to detecting deadlocks introduced by parallelization of RTL blocks that are connected by synthesized interfaces with handshaking protocols in Chapter 6. We conclude this dissertation research and discuss some future research directions in Chapter 7.

Chapter 2

BACKGROUND

2.1 BEHAVIORAL SYNTHESIS

Behavioral synthesis [26] provides a promising approach to dealing with the ever-growing complexity in modern hardware design by generating high-quality RTL implementations from high-level specifications. An early effort was made by researchers at Carnegie Mellon University [51, 58] in the 1970s, which used *instruction set processor specification* as input language. There are open-source behavioral synthesis tools from academia, *e.g.*, SPARK [33], LegUp [17], and Bambu [1]. Commercially available tools from major EDA (Electronic Design Automation) vendors include Cadence's C-to-Silicon Compiler [3], and Cynthesizer [5], Calypto's Catapult HLS [4], NEC's Cyber Workbench [71], Synopsys's Synphony C Compiler [67], and Xilinx's Vivado high-level synthesis [73]. Beside the above C based behavioral synthesis tool, there are also tools use domain-specific input languages, *e.g.*, HDL Coder [52] from MathWorks generates portable, RTL code from MATLAB functions or Simulink models. Bluespec [57] uses Bluespec SystemVerilog as input language.

A behavioral synthesis tool takes an ESL behavioral description of a design (in SystemC or C/C++), together with a library of hardware resources, and generates an RTL implementation [20]. Similar to a compiler, a behavioral synthesis tool first performs lexical, syntax and semantic analysis, and builds an intermediate representation (IR) of the ESL description. A series of transformations is then applied to the IR, which can be categorized into three phases.

- *Compiler transformations* form the first phase. This includes transformations such as dead code elimination, constant propagation, loop unrolling, *etc.*
- *Scheduling transformations* entail computing for each operation the clock cycle for its execution. The clock cycle must account for constraints in hardware resources as well as control and data flow. These transformations include pipelining loops, grouping independent operations for concurrent execution, etc.
- *Resource binding and control synthesis* map each operation to a hardware functional unit, allocates registers for variables used across clock cycles, and generates a finite state machine to implement the schedule.

After these transformations, the design can be represented in RTL. The RTL may be subjected to further manual tweaks.

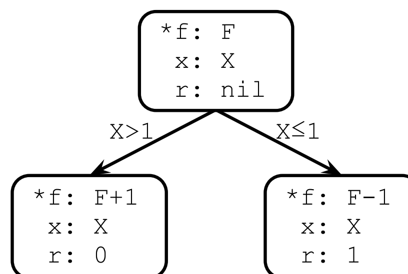
2.2 SYMBOLIC SIMULATION

Symbolic execution [43] is a technique that executes a program with symbolic inputs instead of concrete ones. A symbolic program state includes a statement counter, values of variables and a path condition. Since the inputs are symbolic, the values of variables are expressions over symbolic inputs, and the path condition is a Boolean expression over symbolic inputs. During the execution, a symbolic execution tree is built, with each node representing a program state and each edge representing a state transition condition.

Fig. 2.1 illustrates an example of symbolic execution of a simple program. Fig. 2.1(a) shows a function `foo` written in C, and Fig. 2.1(b) shows its symbolic execution tree. For simplicity, we only show the values of variables (in the box) and path conditions (on the edge). At the beginning of the execution, variable `x` and `*f` take symbolic value `X` and `F`, respectively. Function `foo` has two paths:

```
1 int foo(int *f, int x){
2   if(x>1) {
3     *f = *f + 1;
4     return 0;
5   } else {
6     *f = *f - 1;
7     return 1;
8   }
9 }
```

(a)



(b)

Figure 2.1: A simple function `foo` in C with its symbolic execution tree. (a) Function `foo` in C. (b) Symbolic execution tree of `foo`, where `F` and `X` are symbolic values for `*f` and `x`, `r` denotes the return value, and `nil` denotes that the value is not yet available.

one path increments `f`, and returns 0 with path condition $\texttt{X} > 1$; the other path decrements `f`, and returns 1 with path condition $\texttt{X} \leq 1$.

2.3 FORMAL EQUIVALENCE CHECKING

Formal equivalence checking techniques play an important role in EDA. After synthesize an RTL design to a gate-level implementation, logic equivalence checking tools are used to to prove that the gate-level implementation is functionally equivalent to the RTL design. With decades of research and development, logic equivalence checking has become a mature field. Binary decision diagrams (BDDs) [14] provide canonical representations for Boolean functions, early verification efforts [50] use BDDs for formal logic verification. The idea is to convert two Boolean functions into two BDDs, two functions are equivalent if they have

the same BDD representation. Boolean satisfiability problem (SAT) is to check whether a given propositional formula is satisfiable. There have been research on using SAT for equivalence checking [32]. The basic idea is to convert the equivalence of two circuits problem into satisfiability of a propositional formula, and leverage the state-of-the-art SAT solver to solver the formula.

Our research leverages the success of logic equivalence checking. We borrow the idea of logic equivalence checking of RTL designs and their gate-level implementations to check the equivalence of the high-level (`C/C++`) designs and the generated RTL implementations in behavioral synthesis.

Chapter 3

FRONT-END COMPILER TRANSFORMATION CHECKING

Behavioral synthesis tools apply a sequence of transformations to the input design before scheduling. We check front-end by decomposing it into a sequence of checks, one for each transformation applied. The key observation is that even if the transformation implementations may be closed-source, it is still possible to obtain from most tools the IRs after application of each front-end transformation. Thus, an sequential equivalence checking (SEC) methodology was developed to compare each pair of consecutive IRs.

3.1 NOTATIONS AND DEFINITIONS

Let P be a program, V be the set of variables of P, and $V_{\mathcal{O}} \subseteq V$ be the set of *observable variables*. Intuitively, variables that we can observe during the execution of P are called observable variables; we assume that the $V_{\mathcal{O}}$ includes the input, output, and global variables.

Definition 3.1 (State)**.** A state $s \triangleq \{\langle v, u\rangle \mid v \in V, u \text{ is the value of } v\}$ of a program P is the set of variables in P with their values.

Definition 3.2 (Observable State)**.** An *observable state* $s_{\mathcal{O}}$ at state s of a program P, denoted by $s_{\mathcal{O}}(s)$, is a projection of s, where variables in $s_{\mathcal{O}}$ are restricted to observable variables in program P.

Remark 3.1. *We leave the domains for the values of variables undefined for this presentation, but assume that they can be determined from the context. Also, we*

assume that the domain can be both concrete or symbolic; this permits us to use the same notation for both concrete and symbolic states. For simplicity, we use $s[v]$ to denote the value, either concrete or symbolic, of variable v in state s.

Definition 3.3 (Path)**.** A *path* $\pi \triangleq s_0, c_1, s_1, c_2, s_2, \ldots, c_n, s_n$ of a program is an alternating sequence of states and state transition conditions, starting from an *initial state* s_0 and ending with a *terminal state* s_n, where c_i is the state transition condition (Boolean expression over program variables) from s_{i-1} to s_i for all $1 \leq i \leq n$.

Definition 3.4 (Path Condition)**.** Let $\pi \triangleq s_0, c_1, s_1, c_2, s_2, \ldots, s_{n-1}, c_n, s_n$ be a path of a program P. The *path condition* $pc \triangleq \bigwedge_{i=1}^{n} c_i$ of path π is a conjunction of all transition conditions on π. We use $\pi[pc]$ to denote the path condition of π.

Definition 3.5 (Path Compatibility)**.** Given two programs S and T with the same set of observable variables $V_{\mathcal{O}}$, let π be a path of S with initial state s_0 and path condition pc, and π' be a path with initial state s'_0 and path condition pc' of T. We say π and π' are *compatible* if $s_{\mathcal{O}}(s_0) = s_{\mathcal{O}}(s'_0)$ and $pc \wedge pc'$ is satisfiable. Paths π and π' are called a *compatible path pair* of S and T.

Definition 3.6 (Path Equivalence)**.** Let π be a path of program S with terminal state s_n, and π' be a path of program T with terminal state s'_m, and suppose that programs S and T have the same set of observable variables $V_{\mathcal{O}}$. We say path π and π' are *equivalent*, denoted by $\pi \sim \pi'$, if π and π' are compatible, and for each variable $v \in V_{\mathcal{O}}$, $s_n[v] = s'_m[v]$.

Informally, two paths are equivalent if they are compatible, and they have the same observable state at their terminal states.

Definition 3.7 (Program Equivalence)**.** Let S and T be two programs, we say that program S and T are *equivalent*, denoted by $S \sim T$, if every compatible path pair of S and T has the same observable state at their terminal states. Formally let

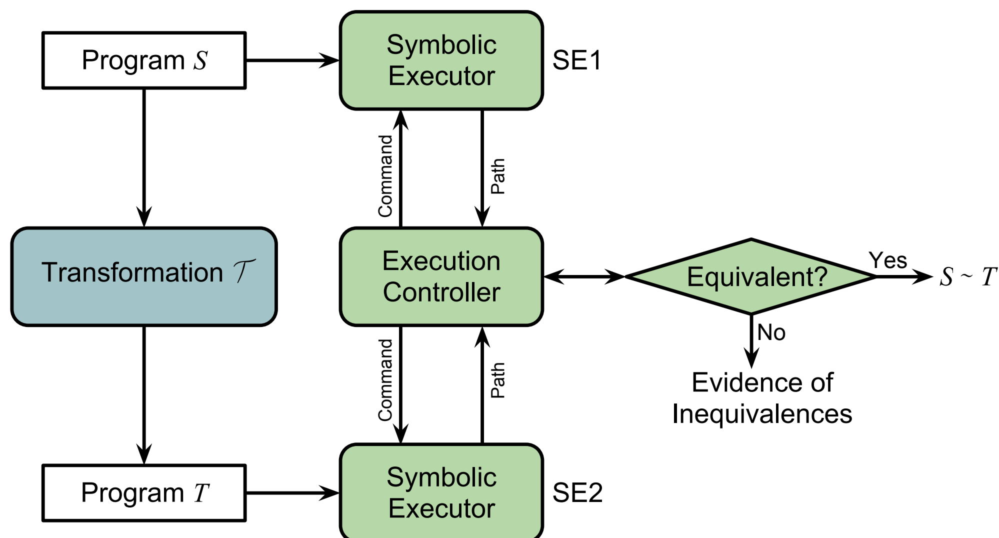


Figure 3.1: Framework of checking equivalence between program S and T, which are the input and output of transformation $\mathcal{T}$ respectively.

$Paths(S)$ and $Paths(T)$ be all paths of program S and T respectively, program S and T are equivalent if for each path $\pi \in Paths(S)$ and every path $\pi' \in Paths(T)$ that is compatible with π, π is equivalent to π'.

We define the correctness of a transformation by the equivalence of the observable behavior of the source program S and the target program T. Informally S and T are equivalent if fed the same inputs to both programs, they produce the same output, and have the same effect on the environment (modification to global variables) when terminating.

Definition 3.8 (Transformation Correctness)**.** Let $\mathcal{T}$ be a transformation which takes a source program S as input and produces a target program T as the output. We say $\mathcal{T}$ is a *correct* transformation on program S if $S \sim T$.

3.2 EQUIVALENCE CHECKING FRAMEWORK

As shown in Fig. 3.1, suppose a transformation $\mathcal{T}$ takes a program S as input and generates a program T as output. We validate the correctness of transformation $\mathcal{T}$ when applied to S by checking whether T is equivalent to S. According to Definition 3.7, we need to prove that S and T have the same observable state at their terminal states for all compatible path pairs.

As a pedagogical simplification, assume that all the paths in S and T are enumerable. Also assume that S and T have the same function signature and global variables; this assumption does not limit our approach because compiler transformations usually do not change function signature unless there are parameters that are irrelevant or unused in the function body, which can be easily detected. Finally, assume that for each observable variable of S we can find the corresponding variable for T and *vice versa*. We then proceed as follows. We assign the same symbols to the input and non-constant global variables of S and T, then symbolically execute them. After enumerating all paths of S and T, for each compatible path pair π in S and π' in T, we check whether π and π' have the same observable behavior; this check is done by an SMT (satisfiability modulo theories) solver by checking the equality between the symbolic expressions of the (symbolic) values of the observable variables.

Algorithm 1 provides a high-level description of our approach. Function CHECK-EQUIVALENCE takes two programs S and T as arguments. Subroutine SYMBOLIZE-INPUTS creates symbols for inputs of S and T. Subroutine SYM-EXE symbolically executes S with symbolic inputs, and collects all paths of S. For each path π with path condition $\pi[pc]$ of S, subroutine GET-OBSERVABLE-STATE collects the observable state s_O corresponding to the terminal state in path π. Subroutine SYM-EXE symbolically executes T with the same symbolic inputs under condition $\pi[pc]$, and collects all paths of T. Since all paths found in T are under the condition

Algorithm 1: CHECK-EQUIVALENCE(S, T)

```
1  s_I ← SYMBOLIZE-INPUTS(S, T)                          ▷ Symbolize inputs
2  Π ← SYM-EXE(S, s_I, nil)                               ▷ Symbolically Execute S
3  foreach π ∈ Π do
4      s_O ← GET-OBSERVABLE-STATE(π)
5      Π′ ← SYM-EXE(T, s_I, π[pc])
6      foreach π′ ∈ Π′ do
7          s′_O ← GET-OBSERVABLE-STATE(π′)
8          if not CMP-STATE(s_O, s′_O, π′[pc]) then
9              print ⟨s_O, s′_O⟩                          ▷ Report inequivalences
10             return false
11 return true
```

$\pi[pc]$, therefore they are all compatible path with π. For each path π' of T found under the condition of $\pi[pc]$, subroutine CMP-STATE checks if π and π' have the same observable state at termination. If the observable states are not equal, the algorithm reports the inequivalences, otherwise it proceeds until all paths of S and T are checked.

Fig. 2.1 and Fig.3.2 show two programs `foo` and `bar` which are defined in C and their independent symbolic execution trees. Before execution, `foo` and `bar` have the same symbolic input $\{\langle *\mathtt{f}, \mathtt{F}\rangle, \langle \mathtt{x}, \mathtt{X}\rangle\}$, where `F` and `X` are symbolic values of variables `*f` and `x`, respectively. Function `foo` has two paths, the final observable states are $\{\langle *\mathtt{f}, \mathtt{F}+1\rangle, \langle \mathtt{x}, \mathtt{X}\rangle, \langle \mathtt{r}, 0\rangle\}$ and $\{\langle *\mathtt{f}, \mathtt{F}-1\rangle, \langle \mathtt{x}, \mathtt{X}\rangle, \langle \mathtt{r}, 1\rangle\}$, with conditions $\mathtt{X} > 1$ and $\mathtt{X} \leq 1$, respectively. Similarly, the two final observable states of function `bar` are $\{\langle *\mathtt{f}, \mathtt{F}+1\rangle, \langle \mathtt{x}, \mathtt{X}\rangle, \langle \mathtt{r}, 1\rangle\}$ and $\{\langle *\mathtt{f}, \mathtt{F}-1\rangle, \langle \mathtt{x}, \mathtt{X}\rangle, \langle \mathtt{r}, 1\rangle\}$, with conditions $\mathtt{X} > 3$ and $\mathtt{X} \leq 3$, respectively.

Fig. 3.3 shows the symbolic execution tree when function `bar` is executed under

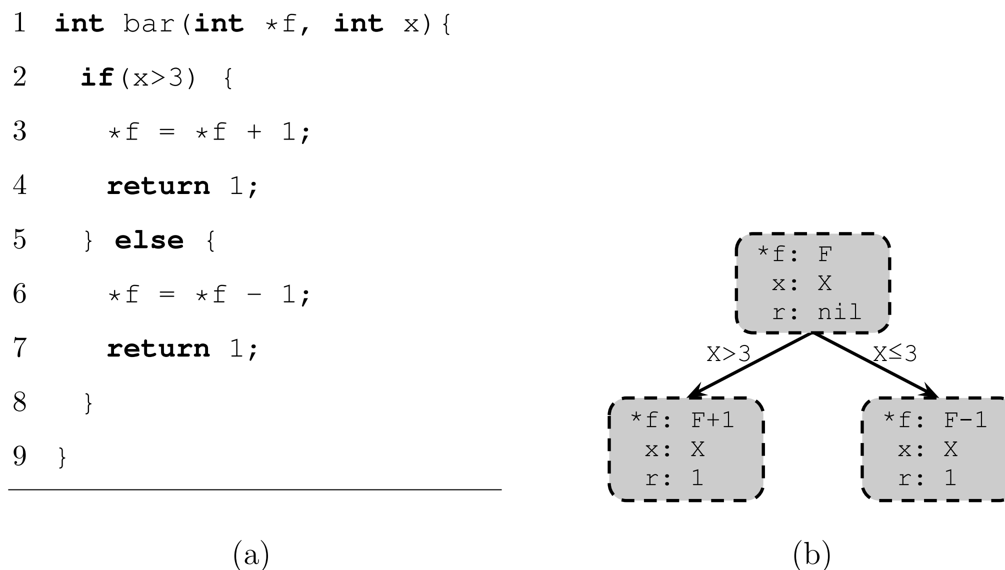


Figure 3.2: A simple function `bar` in C with its symbolic execution tree. (a) Function `bar` in C. (b) Symbolic execution tree of `bar`, where `F` and `X` are symbolic values for `*f` and `x`, `r` denotes the return value, and `nil` denotes that the value is not yet available.

the path condition of function `foo`. In Fig. 3.3, state `s0` is the initial state of function `foo`, states `s3` and `s4` are the initial states of function `bar`, states `s1` and `s2` are terminal states of `foo`, and states `s5`, `s6` and `s7` are terminal states of `bar`. We need to conduct three equivalence checks:

- `s1` vs. `s5`, where the return values are not equivalent;
- `s1` vs. `s6`, where the values of `*f` are not equivalent, and return values are also not equivalent;
- `s2` vs. `s7`, the states are equivalent.

Therefore, our checking algorithm returns that `foo` and `bar` are not equivalent, and reports the inequivalences.

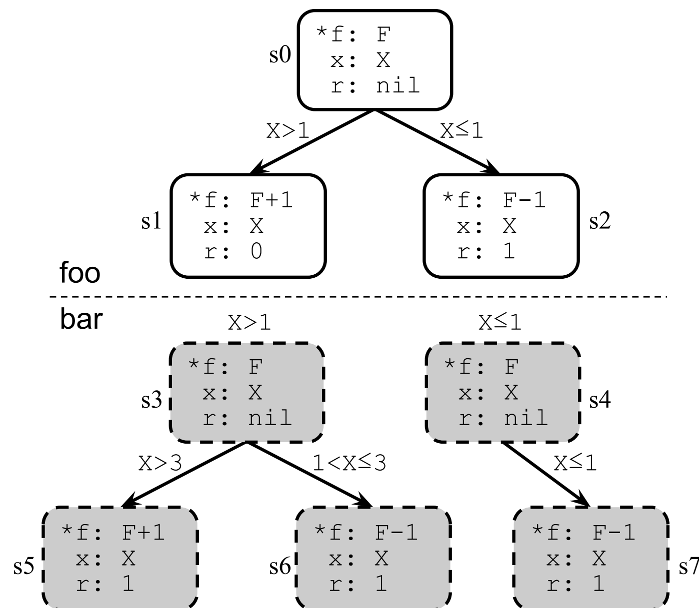


Figure 3.3: Symbolic execution tree of function `foo` and `bar`, where `bar` is executed after `foo`, and based on the execution condition of `foo`.

In summary, the above approach symbolically execute the two IRs (referred to as S and T) and check that each pair of corresponding program paths is equivalent. A program path is uniquely specified by the sequence of branch conditions that must hold for the control flow to execute the instructions in the path. The approach was used on some cryptographic applications. Unfortunately, it does not scale to other practical programs. In particular, it requires effective enumeration of all paths in S and T; in practice, this can lead to *path explosion*. Two key sources of path explosion in practice are subroutine calls and loops. In the next two sections we discuss optimizations to address these problems.

3.3 MODULAR REASONING ACROSS FUNCTIONS

Why do subroutine calls contribute to path explosion? The naïve approach of symbolically executing the program treats each function as if it were inlined: the

function body is symbolically executed at each call site. When a function f having a number of branches in its body (and hence program paths) is invoked many times, each invocation contributes a multiplicative factor to the number of paths explored.

Our approach to address this problem is to develop a compositional approach to symbolic execution [31], which permits equivalence checking on a per-function basis. Suppose functions f and f' invoke functions g and g' respectively. Then (1) we separately check the equivalence of g and g'; and (2) when checking the equivalence of f and f', we replace g and g' with the same uninterpreted function symbols.

Of course the above naïve scheme only works for *side effect* free functions. If g and g' update some global variables or pass-by-reference arguments, then replacing g and g' with the same uninterpreted function on the explicit arguments will be unsound since the effect on the global variable or pass-by-reference arguments is not accounted for. To address this, we use a notion of "extended signature". The idea is to extend the type signature of a function explicitly accounting for the side effects. Let g be a sub-function; we use $\tau = g(\vec{\alpha_v}, \vec{\alpha_r})$ to represent the signature of g, where τ denotes the return value, $\vec{\alpha_v}$ denotes pass-by-value arguments, and $\vec{\alpha_r}$ denotes pass-by-reference arguments. Then, in addition to the function arguments, suppose function g reads globals $\vec{\beta_r}$ and updates globals $\vec{\beta_w}$. The extended type signature of g is:

$$\langle \tau, \vec{\alpha'_r}, \vec{\beta'_w} \rangle = g(\vec{\alpha_v}, \vec{\alpha_r}, \vec{\beta_r}),$$

where $\vec{\alpha'_r}$ and $\vec{\beta'_w}$ are updated versions of $\vec{\alpha_r}$ and $\vec{\beta_w}$ respectively, mimicking the notion that they may be arbitrarily changed by function g. [1]

Fig. 3.4 shows an example of a sub-function with side effects. Function `f` invokes

[1]Pass-by-reference arguments are pointers, and most behavioral synthesis tools restrict the usage of pointers to compile-time determinable ones, which makes this approach works in finding which variable a pointer points to.

```
1  char A;                         // global variable
2  int B;                          // global variable
3  int C[2];                       // global variable
4  void f(int d[4]) {
5   int i = 8;
6   g(i, d);
7  }
8  void g(int x, int y[4]) {
9   B = A + x;                     // side effect on global B
10  C[1] = C[0] + x;               // side effect on global C
11  y[1] = y[0] + x;               // side effect on arguments
12 }
```

Figure 3.4: Global variable usage with sub-function call example.

function `g` which updates the globals `B` and `C`, and pass-by-reference argument `y`.

Extended signatures are exploited to replace function calls with uninterpreted functions symbols. Suppose that function `g` has been certified; when certifying function `f`, we replace `g` with an *uninterpreted function* (say $\mathcal{G}$) of four arguments, and the effect of the invocation of `g` on the globals (`B` and `C`) and argument `y` is given by:

$$\langle \mathtt{d}, \mathtt{B}, \mathtt{C} \rangle = \mathcal{G}(\mathtt{i}, \mathtt{d}, \mathtt{A}, \mathtt{C}).$$

Since each invocation of sub-functions is replaced by an uninterpreted function symbol, we alleviate path-explosion problem introduced by subroutine calls. It is worth noting that this can generate false alarms, and we need to take special care to handle common false negatives. We discuss this issue when describing our experiments.

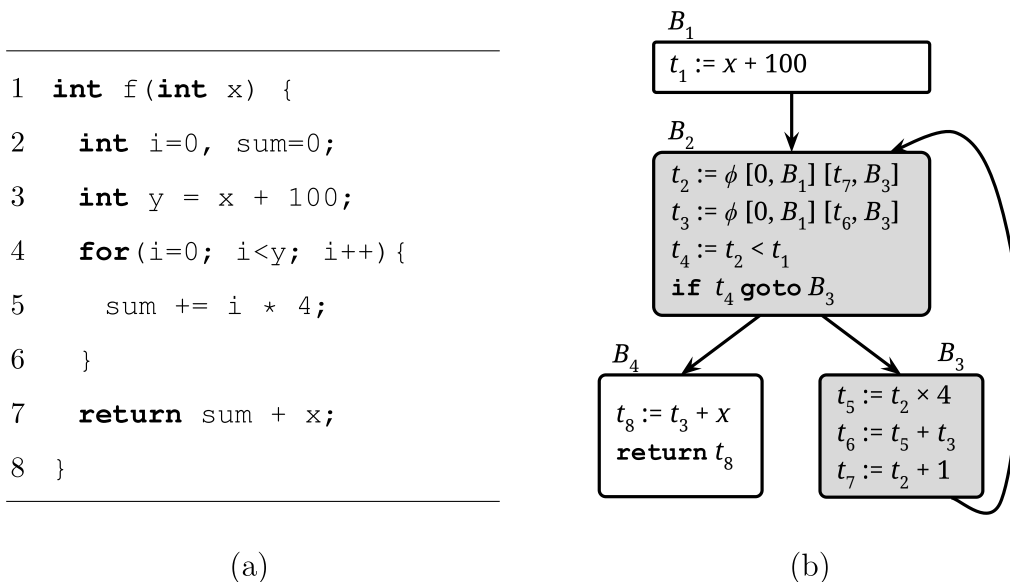


Figure 3.5: A simple function with a loop in C and its IR. (a) Function `f` has an unbounded `for` loop. (b) The IR of `f`, with boxes representing basic blocks, and arrows representing control flow. Control flow merge is implemented via ϕ-instructions in basic block B_2.

3.4 HANDLING LOOPS

Loops are the second major contributors to path explosion (and, in case of unbounded ones, non-termination) in symbolic simulation of software programs. The reason is that symbolic simulation of a loop induces (at least) two branches for each loop iteration simulated: (1) the branch where the loop test holds (and hence the body is executed) and (2) the branch where the test is false.[2]

We handle equivalence of loops by an approach called *cut-loop optimization*. Our approach borrows ideas from a corresponding one for back-end SEC between high-level IR and RTL [36], and is an adaptation of classic inductive assertions

[2]The branching may be limited if the value of the loop test can be computed concretely during symbolic execution of the loop. However, this is not possible for most non-trivial loops in practice.

approach [29, 38] to program equivalences. The idea is to “cut” the loop, which reduces equivalence of loop execution to equivalence checks at entry, body, and exit. To illustrate the idea, consider the example in Fig. 3.5. Fig. 3.5 (a) shows a simple unbounded loop in C. Fig. 3.5 (b) shows the IR of function `f`.[3] If the input `x` is symbolic, then the symbolic expansion of the loop (and hence the symbolic execution of `f`) will not terminate.[4] We assume loops are in *natural loop* form in the IRs. A *natural loop* must have a single entry point (header) and at least one *back edge* leading the control flow from the loop body back to the loop header. Fig. 3.5 (b) is an example of a natural loop, where B_2 is the loop header, and edge (B_3, B_2) is the back edge.

Back-end SEC [36] exploits mappings of variables provided by behavioral synthesis tool. For simplicity, suppose each basic block is a scheduling step in Fig. 3.5 (b). At the end of execution of each scheduling step, we check the equivalence of all mapped variables. If they are equivalent, we replace every pair of mapped variables with the same symbolic symbol (with *cut-point* optimization). Suppose the loop is entered from B_1, after executing of B_3, variables t_5, t_6 and t_7 are checked equivalence with their mapped variables in RTL, and then will be replaced with symbolic variables. Till this point, we checked the first iteration of the loop. The subsequent iterations of the loop are all led by the back edge. Since we made t_5, t_6 and t_7 symbolic, executing the subsequent iterations once will cover all possible cases. All we need to do is to avoid multiple entrance of a loop through the same back edge.

---

[3]Control flow merge is implemented via ϕ-instructions in basic block B_2. A ϕ-instruction `v` $= \phi\,[\alpha, B_i]\,[\beta, B_j]$ in basic block B means that `v` has the value α if B is reached from B_i, and β if reached from B_j.

[4]Technically, symbolic simulation can terminate when a fixpoint is reached, *i.e.*, when all reachable states have been explored. But achieving such fixpoint requires the restriction that all the variable types are finite, as well as an expensive fixpoint computation through symbolic simulation.

However, variables mappings between two IRs are not available during front-end transformations. Suppose a loop consists of a list of basic blocks; then we must identify a minimum set of variables that need to be mapped between the two IRs. In particular, we need to identify mappings for loop-carried variables. The reason is that we want to make them symbolic, so that we only need to execute the loop once through each back edge (see below).

We achieve the above through *use-definition chains* analysis [10]. In the example shown in Fig. 3.5 (b), loop L consists of basic blocks B_2 and B_3, variable t_6 and t_7 are loop-carried variables. For example, execution of loop L is done as follows:

1. Loop L is entered from B_1:
   - path 1: $B_1 \rightarrow B_2 \rightarrow B_4$: we check the equivalence of return variable t_8;
   - path 2: $B_1 \rightarrow B_2 \rightarrow B_3$: we check the equivalence of loop-carried variables t_6 and t_7, and make them symbolic afterwards.
2. Loop L is entered from B_3 through the back edge:
   - path 1: $B_3 \rightarrow B_2 \rightarrow B_4$: we check the equivalence of return variable t_8;
   - path 2: $B_3 \rightarrow B_2 \rightarrow B_3$: we check the equivalence of loop-carried variables t_6 and t_7, and terminate L.

The above approach requires detecting the loop structure (*e.g.*, loop header, exit, back edge, etc.) in the IRs. Once the loop-carried variables are identified, equivalence of loop computation can be verified by checking the first iteration (entered from entry) and one subsequent symbolic iteration (entered through back edge) for each back edge; since we made loop-carried variables symbolic after first iteration, this covers all possible cases for the subsequent iterations. The sufficiency of these checks was mechanically proven in previous work using the ACL2 theorem prover [63].

Finally, cut-loop requires that when the corresponding loops in the two IRs being compared have the same structure, and perform equivalent computation at each iteration, *e.g.*, it is inapplicable if the transformation entails partial loop unrolling. However, as our experiments indicate, most behavioral synthesis transformations are structure-preserving, making the optimization widely applicable.

3.5 EXPERIMENTAL RESULTS

Table 3.1: Summary of CHStone Benchmark for Equivalence Checking of Front-end Compiler Transformations

App. Domain	Design	Lines of C Code	Lines of RTL	# of Functions
Arithmetic	`DFADD`	542	12933	17
	`DFDIV`	452	10948	19
	`DFMUL`	392	7100	16
	`DFSIN`	772	22949	31
Microprocessor	`MIPS`	256	7237	1
Media Processing	`ADPCM`	521	33706	15
	`GSM`	388	22816	12
	`JPEG`	1031	53584	30
	`MOTION`	414	13770	13
Security	`AES`	699	40014	11
	`BLOWFISH`	1241	23490	6
	`SHA`	1284	12491	8

We applied our framework to CHStone [37], a publicly available behavioral synthesis benchmark suite containing 12 ESL designs (in C). We used LegUp [17] to synthesize these designs. We conducted our experiments on a workstation with Debian 7.1 running on a 2.93 GHz Intel Xeon X3470 processor with 8 GB of

Table 3.2: Summary of Evaluation on CHStone Benchmark for Equivalence Checking of Front-end Compiler Transformations

Design	# of Checked Transformations	# of Successful Checks	Success Rate (%)	Avg. Time (s)	Memory (MB)
DFADD	62	62	100.00	0.78	159.86
DFDIV	62	78	79.49	0.95	161.22
DFMUL	47	48	97.92	0.93	155.93
DFSIN	113	115	98.26	0.88	187.70
MIPS	10	13	76.92	2.32	15.01
ADPCM	69	101	68.32	3.33	123.85
GSM	53	86	61.63	0.26	122.94
JPEG	158	237	66.67	1.55	694.76
MOTION	59	74	79.73	0.49	52.50
AES	67	85	78.82	4.22	120.56
BLOWFISH	27	48	56.25	3.28	93.58
SHA	36	61	59.02	0.05	106.14

memory. We focused on intra-procedural transformations. The experiments were run with a cutoff time of 90 seconds: certifications taking longer than this time are classified as failures. The reason for this cutoff is that in our experience, most successful transformation certifications that complete in any reasonable time finish within a few seconds of this size; if symbolic execution takes more than 90 seconds, it is unlikely to finish. Thus we believe that the impact of making the cutoff longer on the number of successful transformations will be insignificant. Our tool supports a number of SMT solvers. The results on this benchmark use Z3 [24] since it outperforms others.

Table 3.1 and 3.2 show the statistics of the experiments, *e.g.*, the number

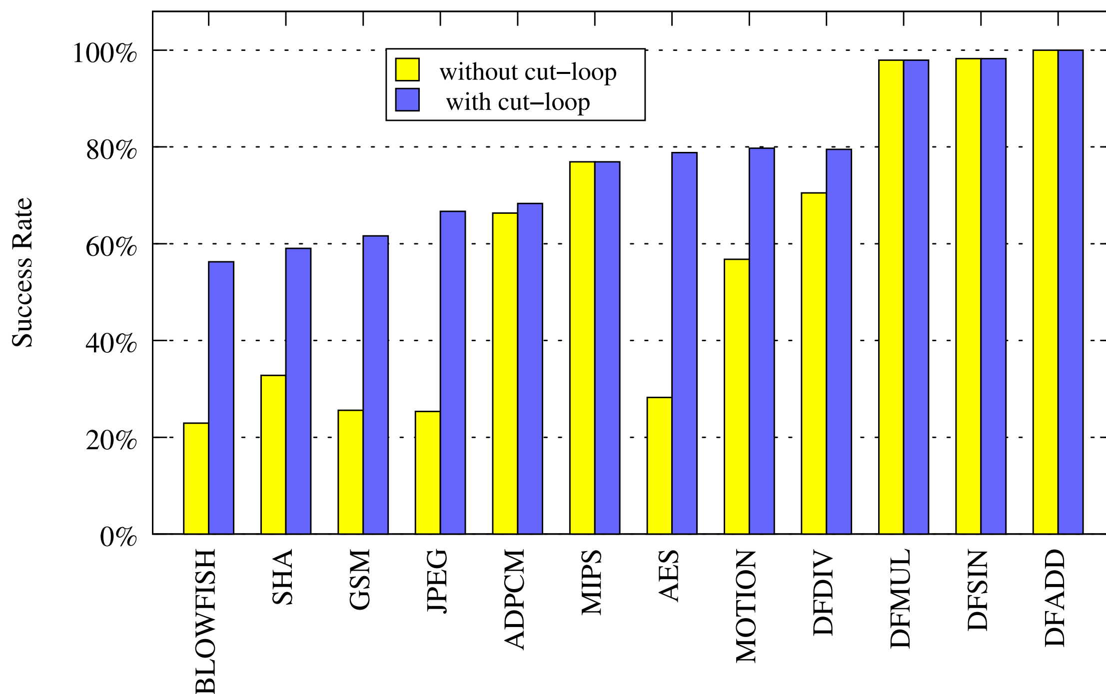


Figure 3.6: Comparison of success rate on designs of CHStone benchmark without and with cut-loop optimization. The x axis is ordered by the success rate with cut-loop.

of transformations applied by the synthesis tool,[5] the number of transformations checked successfully, as well as time and memory usages. In all successful cases, time and memory usages are modest. With compositional execution and cut-loop optimization, we successfully validated 75.69 percent of transformations (763 out of 1008).

Fig. 3.6 compares the success without and with cut-loop optimization. Without cut-loop, we can validate only 52.88 percent of transformations (533 out of 1008). Cut-loop optimization provides an improvement of 22 percent. The improvement is most significant for `AES`, `JPEG`, `GSM`, and `BLOWFISH` since they have more loops. The transformations that fail certification (about 25 percent) typically do so for

[5]Some transformations are applied more than once.

two reasons: (1) the transformation changes loop structure thus makes cut-loop inapplicable; or (2) symbolic expressions of corresponding variables in source and target programs become too complex, causing blow-up of the SMT solver.

Since we focus on intra-procedural transformations, we check the designs compositionally; this may sometimes introduce subtle false alarms. False alarms can arise in surprising ways, *e.g.*, the extended signature of a sub-function is different in the source and target programs. As an example [6], suppose that a function `foo` has a sub-function `legup_memcpy` that is invoked as follows.

```
r = legup_memcpy(a, b, c)
```

This sub-function has a return value, but the value is never used subsequently. Similarly, suppose that a function `bar` invokes the same sub-function `legup_memcpy` as follows:

```
legup_memcpy(a, b, c)
```

This invocation treats `legup_memcpy` as if it returns void. The problem is when we compositionally check `foo` and `bar`, we abstract sub-function `legup_memcpy` with uninterpreted function. Since the extended signatures are different for two invocations of the sub-function, (one with a return value and the other one without), a naïve approach will report an inequivalence due to type mismatch. Since the result of return value `r` in `foo` is not used, this inequivalence is a false alarm. However, it can be easily eliminated, by excluding from the extended signature the types of return values that are not subsequently used.

3.6 RELATED WORK

Formal verification of compilers: There has been research on formally proving compiler transformations correct by theorem prover. CompCert [49] is the

[6] This is a real example in function `Fill_Buffer` in `MOTION` design. The transformation is called "Combine Redundant Instructions"

first formally verified compiler. Similar to CompCert, Vellvm [76] project formalizes LLVM's intermediate representation, and develops a framework for reasoning about programs. Ray *et al.* proposed an certification framework for transformations in behavioral synthesis [64]. This framework uses theorem proving to certify high-level transformations. However, using theorem proving to prove all transformations requires enormous manual effort; it also requires knowledge about internal algorithms of each transformation, which is often not available because most behavioral synthesis tools are closed source.

Translation validation: Pnueli *et al.* proposed the notion of *translation validation* [59] for validating the transformations during compilation. Instead of verifying a transformation once and for all, they showed how to generate a proof of correspondence between the source and target programs for each individual transformation. However, it is problematic for the approach to handle programs with more than one loop. Zuck *et al.* extended this approach to support structure-modifying transformations [77]. Necula used symbolic evaluation techniques from proof-carrying code to tackle translation validation [56]. However, this approach only handled transformations where source and target programs have the same branch conditions. Zaks and Pnueli proposed a framework to construct the cross product of the source and target programs [75]. This reduces the problem of checking the equivalence of two programs to verification of a single program. Peggy [69] performed translation validation for the LLVM compiler using *equality saturation*. It built Program Expression Graphs for the source and target programs of a transformation and then reasons about equalities among graph nodes. If output nodes of two programs are shown equal, the two programs are equivalent. To our knowledge, these approaches do not scale to programs of the size we consider in this research. Note that a key reason for the differences in scalability is that the aim of the above line of research is to check the correctness of generic compiler transformations while we focus on the transformations in behavioral synthesis. In particular, the

programs being synthesized can be represented as finite state machines and many language features such as dynamic memory allocation are prohibited.

Symbolic execution based techniques: There has also been recent research on applying symbolic techniques to checking the equivalence of two arbitrary programs. UC-KLEE [62] proposes a smart stub function that invokes two routines that need to be verified for equivalence, and leverages KLEE [15] to symbolically execute the stub function. Upon finding a path, UC-KLEE checks if the two routines behave the same. UC-KLEE enumerates path with best efforts; thus it suffers from path explosion and does not terminate when executing unbounded loops. SYM-DIFF [48] is a symbolic differentiation tool, which symbolically executes two programs with same symbolic inputs, and checks if the two programs have identical outputs. SYM-DIFF handles loops by unrolling them to a user-specified depth; consequently it cannot certify equivalence between programs whose loops iterations are long or controlled by input variables; we can handle such programs with cut-loop optimization.

3.7 SUMMARY

In this Chapter, we have presented a scalable SEC framework to validate the correctness of front-end compiler transformations in behavioral synthesis. We use symbolic execution technique to explore (possibly all) paths of the source and target programs of each transformation. We showed how to ameliorate path explosion and non-termination in symbolic simulation through compositionality and cut-loop optimization. Our framework can fully automatically certify results of more than 75 percent of 1008 transformations employed by a synthesis tool on designs from the CHStone benchmark. We are not aware of any SEC framework that can handle compiler transformations at such diversity and scale.

Our results underline the importance of aligning verification methodology with

the design flow in the development of a scalable verification framework. SEC for behavioral synthesis transformations at the scale achieved here has not been done before because extant tools focused on input/output equivalence between the high-level ESL description and synthesized RTL; such efforts are ineffective because of the high abstraction gap. On the other hand, pre-certified compiler transformation via theorem proving as proposed in previous work [64] was not successful both because of the number and complexity of such transformations and the reluctance of synthesis tool vendors to expose transformation implementation for formal analysis. Our key insights are that (1) design IRs before and after each transformation application can be made available from a commercial synthesis flow even if the transformations themselves are proprietary, and (2) restrictions in program features enforced by behavioral synthesis from the need to eventually generate hardware circuit from the design description make it possible to use "black-box" SEC techniques effectively to certify these IRs. The key take-away from our paper is that once the right verification methodology has been identified, it is possible with insight of the source of verification complexity of the domain to adapt well-known analysis ingredients into an end-to-end certification solution in a complex domain.

One possible argument against our framework is the requirement that IRs after each transformation application be available to the tool. In particular, if the validation is performed by a third party, this requirement may provide exposure to confidential design intellectual property (IP). In practice, we have not seen that to be a problem for two reasons. First, in many industrial contexts, the validation is performed by personnel who have access to the original ESL and RTL designs anyhow (*e.g.*, by a validation group in the same organization that designed the ESL). Second, most extant commercial behavioral synthesis tools already provide the information on IRs; we do not require any additional information to perform our analysis. Nevertheless, the potential of IP leakage is an important one, and we

plan to look at the constraints and data available to third-party evaluators during design certification in future work to determine how our framework can be made usable in that context.

Chapter 4

VALIDATING SCHEDULING TRANSFORMATION

4.1 SCHEDULING TRANSFORMATION

Scheduling is a critical synthesis phase that directly affect the quality of synthesized design in terms of timing, performance, and thermal characteristics [22]. Scheduling transformation involves complex heuristics to ensure that the design being synthesized can meet the timing and resource constraints while preserving control and data dependencies.

To understand the source of resource constraints consider scheduling the following operations:

S1: `x = y * z;`
S2: `p = x + y;`
S3: `w = a * b;`
S4: `q = a + z;`

Assume that the design is not being pipelined, the system has a single multiplier that requires 3 cycles for completion, and two adders. Suppose the scheduling transformation schedules **S1** to start at clock cycle t. Then, since there is data dependency between **S1** and **S2**, the operation **S2** cannot be scheduled before t. Furthermore, since multiplier takes 3 clock cycles, **S2** cannot be scheduled to start before cycle t+3. Finally, since there is only one multiplier, **S3** cannot be scheduled to start before cycle $t+3$ either, although there is no data dependency between **S1**

```
1  void block::thread() {
2    int accu = 0;
3    wait();
4    while(1) {
5      int a = In.read();
6      Out.write(a + accu);
7      wait();
8      int b = In.read();
9      Out.write(b * b);
10     accu = In.read();
11     wait();
12   }
13 }
```

Figure 4.1: An example of a SystemC thread, which has two superstates in the `while` loop.

and **S3** (or **S3** is scheduled at clock cycle t, then **S1** can not be scheduled before cycle $t + 3$). On the other hand, since there are two adders, **S2** and **S4** can be scheduled concurrently.

For untimed `C/C++` designs, scheduling transformation can assign any clock cycle to an I/O operation as long as control/data dependencies and resource constraints as discussed above are met. However, most high-level descriptions of hardware designs also specify partial timing. For example, SystemC designs can have default I/O timing constraints that are usually specified by `wait` statements. Behavioral synthesis tools usually provide the user the flexibility to explore different architectures by picking different I/O scheduling modes [26] as discussed below.

1. *Cycle-Fixed mode*: In this mode, the user explicitly specifies the timing of the I/O operations, and the scheduling transformation cannot change or

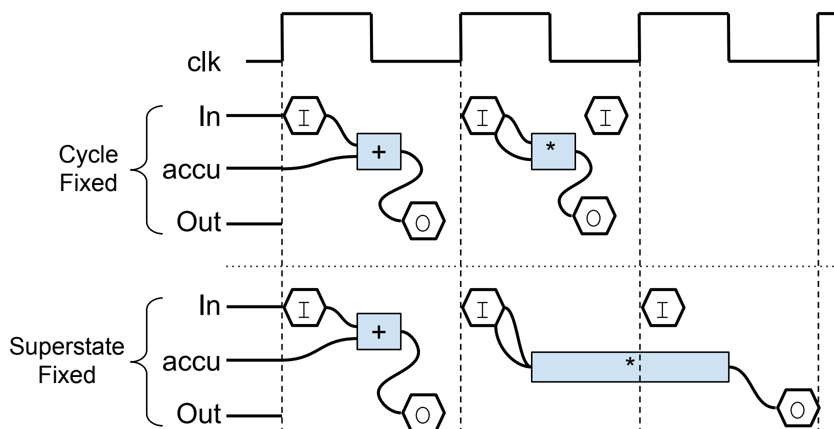


Figure 4.2: Cycle-fixed and superstate-fixed scheduling mode for the `while` loop in thread in Fig. 4.1.

refine this timing. This is applied typically at design interfaces which implement communication protocols (possibly with other external interfaces), and the cycles when data must be read or written is governed by the protocol. Fig. 4.2 shows an example of cycle-fixed scheduling for the thread in Fig. 4.1. Note that the number of states is 2, which is the same as the design before scheduling.

2. *Superstate-Fixed mode*: In this mode, the user specifies `wait` statements. The scheduler comprehends these `wait` statements to be the boundaries of "superstates", which impose constraints on scheduling I/O operations as follows. Informally, a superstate is a sequence of operations, possibly scheduled over multiple clock cycles, with the requirements that (1) no I/O operation in a superstate can be moved across the superstate boundary, and (2) all I/O writes must be scheduled at the last clock cycle assigned to operations in the superstate. Fig. 4.2 shows an example of its use for thread in Fig. 4.1. Here we assume that the multiplication takes two cycles, we can see that the `read` in line 10 and the `write` in line 9 are scheduled to the third clock cycle.

3. *Free-Floating mode*: The scheduling transformation can assign any I/O operation to any clock cycle (possibly switching their program order), even add or delete clock cycles, as long as control/data dependencies are maintained.

4.2 FORMALIZATION

In order to formalize the validation requirement to certify scheduling transformations, we need a notion of correspondence between IRs before and after scheduling. In this section, we develop the formalization of this notion. Note that the notion we require for certifying a specific scheduling application depends on the scheduling mode for the application.

Assumptions and Conventions. Our notion of correspondence relates executions of IRs before and after scheduling. Consequently, it depends on a formal semantics of IRs. Our formalization uses the Control/Data Flow Graph (CDFG) defined below. It is widely used as design representations in research on compiler and intermediate languages. Our formalization assumes that (1) the set V_o of operations in an IR is a subset of a fixed set O of all operations; and (2) all operations in O are defined through operational semantics over abstract machine states. We also assume the IR to be naturally decomposed into a collection of basic blocks V_b. These assumptions are standard in formalization of control constructs for programming language semantics. Furthermore, O is assumed to contain standard variable *read* and *write* operations with the usual meaning, and a *wait* operation that specifies a transition in the operational model with no effect on the underlying machine state. Given a set V of operations, we define the *active subset* of V, denoted by $V[N]$ to be the subset of V excluding all wait statements. Finally, we restrict operations in O to be in Single Static Assignment (SSA) form. This is also standard for IRs generated by compiler transformation, *e.g.*, LLVM imposes this restriction. Consequently, for each operation $o \in V_o$ there is a unique basic

block in V_b containing o. Control and data flows are determined by data dependencies via usual read-after-write paradigm: for $o_1, o_2 \in O$, o_2 depends on o_1 if (1) o_1 appears before o_2 in the program order, and (2) o_2 reads the value of some variable written by o_1. Following conventions from program analysis, the control flow graph of an IR is a directed graph $G_C \triangleq (V_b, E_c)$ where an edge $e \in E_c$ from basic block b_0 to b_1 represents a control dependency of b_1 on b_0, and the data flow graph is the directed graph $G_D \triangleq (V_o, E_d)$ where an edge $e \in E_d$ from operation o_1 to o_2 represents a data dependency of o_2 on o_1. For convenience we assume that for any wait operation w and any write operation o preceding w in the IR, there is an edge from o to w in G_D; correspondingly, for any read or write operation o' following w, there is an edge from w to o'.

Definition 4.1 (CDFG)**.** The CDFG is a triple $G \triangleq (G_C, G_D, R)$, where $G_c \triangleq (V_b, E_c)$ is a control flow graph, $G_D \triangleq (V_o, E_d)$ is a data flow graph, and R is a mapping $R : V_o \rightarrow V_b$.

Informally, for each operation $o \in V_o$, $R(o)$ represents the basic block for o. The mapping is well-defined by the uniqueness assumption.

The goal of scheduling transformation is to assign to each operation a clock cycle of execution. For this paper, we assume that the set of operations does not change due to scheduling, and the control/data flow remains unaffected. This is justified since operations are typically modified by compiler transformation before scheduling, which can be certified through a separate equivalence checking technique as achieved by previous work [74]. Furthermore, control and data flow in behavioral synthesis are typically modified by either compiler transformations or pipelining, which can also be separately certified [34].

We formalize the timing associated with an operation execution with the notion of a *state transition partition* (STP) defined below. In the following definition, it is convenient to interpret the pair (P_i, τ_i) as the directive that (active) operations

in P_i are scheduled at clock cycle τ_i.

Definition 4.2 (STP)**.** Let V_o be a set of operations. A *state transition partition* of V_o is a finite set of pairs $\{(P_i, \tau_i) : i = 1 \ldots k\}$, where each P_i is a sequence of operations over $V_o[N]$ and the following conditions hold:

1. $\bigcup_{i=1}^{k} P_i = V_o[N]$;

2. $P_i \cap P_j = \emptyset$ for $i \neq j$;

3. $\tau_i \in \mathbb{N}$ with $\tau_i \neq \tau_j$ for $i \neq j$.

If the pair (P, τ) is a member of STP S then for any operation $o \in P$ we represent τ as $\tau_S[o]$ and P as $P_S[o]$, dropping the subscript when there is no ambiguity.

STP can be viewed as a grouping of active operations in V_o by associating them to a clock cycle for execution. Note that each partition is specified as a sequence rather than a set. The reason is that due to control/data dependencies one cannot execute all the operations together even if all of them can be completed (perhaps sequentially) within one clock cycle. It is convenient to view each partition P_i as a sequence requiring that if an operation o appears before o' then o' cannot be scheduled for execution before o. We utilize this restriction in defining trace compatibility below, which relates operations in STP with the control/data flow requirements specified by a CDFG. Informally, we want that the operation scheduling must respect the data and control dependencies in the CDFG, *i.e.*, an operation o can be scheduled at cycle τ only after any operations o depends on have completed, either in a previous cycle or earlier in the same cycle.

Definition 4.3 (Operation Precedence)**.** Given an STP S over a set of operations V_o, and two operations $o_1, o_2 \in V_o$, we say o_2 *follows* o_1 in V_o if either (1) $\tau[o_2] > \tau[o_1]$, or (2) $\tau[o_1] = \tau[o_2]$ and o_2 appears after o_1 in $P[o_1]$.

Definition 4.4 (Trace Compatibility)**.** Let $G \triangleq (G_C, G_D, R)$ be a CDFG over the set of operations V_o and basic blocks V_b, and let $S \triangleq \{(P, \tau)\}$ be an STP over V_o. We say that S is *compatible* with G is the following conditions hold for each pair of operations o_i and o_j in $V_o[N]$:

1. If there is a path from o_i to o_j in G_D then then o_j follows o_i in S.

2. If there is a path from $R(o_i)$ to $R(o_j)$ in G_C then o_j follows o_i in S.

In addition to respecting control/data flow requirements from CDFG specified by the definition of Trace Compatibility, scheduling must also satisfy the I/O restrictions as specified by the scheduling mode. Formally, we capture the I/O restrictions for each scheduling mode by further restricting for I/O operations the timing constraints.

The most "rigid" scheduling mode is cycle-fixed. We formalize its requirement in the following definition. Informally, the definition requires that the I/O operations be scheduled strictly following the timing constraints specified by the user.

Definition 4.5 (Valid Cycle-Fixed Schedule)**.** Let $S \triangleq \{(P_i, \tau_i), i = 1, \ldots, k\}$ over an operation set V_o and $G \triangleq (G_C, G_D, R)$ be a CDFG. We say that P is a *valid cycle-fixed schedule* with respect to G if S is compatible with G, and the following additional condition holds:

> Let o_1 and o_2 be two read or write operations such that $R(o_1) = R(o_2)$. Suppose that there is a path Π in G_D from o_1 to o_2 that has n `wait` operations. Then $\tau[o_2] = \tau[o_1] + n$.

The scheduling requirements for the superstate-fixed mode are the most elaborate. We formalize them below.

Definition 4.6 (Valid Superstate-Fixed Schedule)**.** Let $S \triangleq \{(P_i, \tau_i), i = 1, \ldots, k\}$ over an operation set V_o and $G \triangleq (G_C, G_D, R)$ be a CDFG. We say that P is a

valid superstate-fixed schedule with respect to G if S is compatible with G, and additional timing conditions hold which are specified as follows. Let o_i and o_j be two read or write operations such that $R(o_i) = R(o_j)$. Suppose that there is a path Π in G_D from o_i to o_j that has n `wait` operations. Then $\tau[o_j] \geq \tau[o_i] + n$. In addition, let o_1, o_2, o_3 be operations such that $R(o_1) = R(o_2) = R(o_3)$, o_1 and o_2 are write operations, and o_3 is a `wait` operation. Suppose that there are paths Π_1 and Π_2 in G_D from o_1 to o_3 and o_2 to o_3 such that there is no intermediate wait operation. Then:

1. $P[o_1] = P[o_2]$.

2. Let o be any operation such that $R(o) = R(o_1)$ and there is a path Π from o to o_1 (resp., o_2) in G_D. Then $\tau[o] \leq \tau[o_1]$ (resp., $\tau[o] \leq \tau[o_2]$).

3. Let o be any operation such that either (1) there is a path Π from o_1 (resp., o_2) to o in G_D, or (2) there is a path Π' from $R(o_1)$ to $R(o)$ in G_C (resp., o_2). Then $\tau[o_1] \leq \tau[o]$ and $\tau[o_2] \leq \tau[o]$.

We ensure that the scheduling does not "squeeze" I/O operations by removing clock cycles. In addition, conditions 1 – 3 ensure that any write operation is scheduled in the last before any user-provided wait operation. Note that unlike the cycle-fixed mode, scheduling can introduce additional wait operations in this mode, thereby "stretching" operation scheduling to more cycles than specified by user-provided wait operations.

Finally, for free-floating mode, since there is no additional restriction on I/O operations, a valid schedule is one that satisfies Trace Compatibility.

4.3 VALIDATION APPROACH

4.3.1 Validating Trace Compatibility

Let G be the CDFG of a design and S be the STP after scheduling for a set V_o of operations. Our approach to control/data dependency checking (and hence free-floating mode scheduling) is to first define a *dependency graph* G_Δ, consolidating the control and data dependencies. The dependency graph $G_\Delta = (V_\Delta, E_\Delta)$ where V_Δ is a set of operations, E_Δ is a tuple $\langle o_i, o_j, \mathcal{C} \rangle$, which is interpreted to mean that operation o_j depends on o_i under condition $\mathcal{C}$. Condition $\mathcal{C}$ is a conjunction of Boolean variables. Note that dependency graph G_Δ not only captures the data dependencies of a design, but also includes control dependencies through the condition encoded in each edge. Constructing the graph requires a traversal of CDFG G, identifying for each operation $o \in V_o$, the condition under which o is executed. This can be done efficiently using *def-use* chain analysis [10], exploiting the SSA form of the operations: since the left-hand-side of every assignment is unique we can trivially identify variable dependency chains. Finally, to check if the control/data dependencies are satisfied in S, it is sufficient that for each pair of operations o_i and o_j, o_j follows o_i in S under condition $\mathcal{C}$.

According to Definition 4.4, S is *compatible* with G if control and data dependencies are preserved in S. Therefore we can validate the trace compatibility by comparing the dependency graphs of S and G.

4.3.2 Validating I/O Timing

The I/O timing is important for partially timed designs (*e.g.* `wait` statements explicitly specified in SystemC). Even for untimed C/C++ designs, some behavioral synthesis tools allow the user to specify a *protocol region* of design to instruct the scheduling transformation to preserve the timing of the I/Os within the region. Within the protocol region, the scheduling transformation will not re-order the

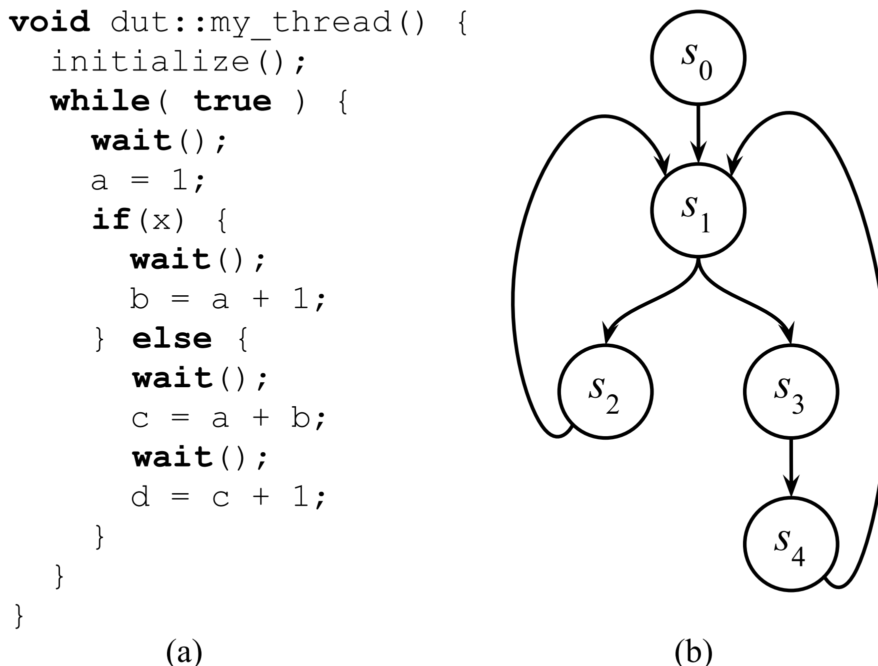


Figure 4.3: Extract *superstates* of a thread in SystemC. (a) `my_thread` is a thread of module `dut`, `wait` statements are the boundary of superstates in SystemC. (b) Superstates and their transitions of `my_thread`.

I/O operations, and will not insert `wait` statements. The `wait` statements specify the boundary of superstates.

For Cycle-Fixed and Superstate-Fixed modes, we need to additionally validate the I/O timing requirements. In order to check the timing requirements, it will be convenient to call the partitions in the STP S to be *states* and the set of operations between any two user-specified `wait` operations in a CDFG G to be *superstates*. Fig. 4.3 (a) shows an example of a module `dut` in SystemC, where `my_thread` is a thread of `dut`. Fig. 4.3 (b) shows the superstates and their transitions in `my_thread`.

Let an execution trace $\pi = [s^s_1, s^s_2, \ldots, s^s_i, \ldots]$ of a CDFG $G \triangleq (G_C, G_D, R)$ be a sequence execution of superstates following control flow in G_C. Let an execution trace $\pi' = [s^g_1, s^g_2, \ldots, s^g_j, \ldots]$ of an STP S be a sequence of operation segments,

Algorithm 2: CHECK-CYCLE-FIXED-MODE(G, S)

```
1  ss ← BUILD-SUPERSTATES(G)
2  T1 ← COMPUTE-TRACES(ss)
3  T2 ← COMPUTE-TRACES(S)
4  foreach π = [s1^s, s2^s, ..., sm^s] ∈ T1 do
5      C ← EXTRACT-TRACE-COND(π)
6      [s1^g, s2^g, ..., sn^g] ← FIND-TRACE(T2, C)
7      assert m = n                    ▷ have the same number of cycles
8      for i ← 0 to m do
9          assert HAS-1-1-MAPPING(F_rw(si^s), F_rw(si^g))
10 return true
```

such that operations in each segment belong to the same partition. Note that traces π and π' can be infinite due to loops structures. Since we have already checked the control and data dependencies of G and S in Section 4.3.1, we know the loop structures are preserved by the scheduling transformation. Thus, when computing the traces of G and S, we can break loop back-edges temporarily. As a result, traces in G and S will be finite.

Let $\mathcal{F}_{rw}$ be a projection function, which takes a superstate s_i^s or operation segment s_i^g, outputs only *read* and *write* operations. Similarly, let $\mathcal{F}_w$ be another projection function, which only outputs *write* operations.

We are now ready to formally specify our algorithm for Cycle-Fixed and Superstate-Fixed modes:

From the requirements of Cycle-Fixed mode, the number of superstates after scheduling must be equal to the number of scheduled states. Algorithm 2 checks if STP S is a valid Cycle-Fixed scheduling of CDFG G. Function BUILD-SUPERSTATES build superstates ss from G. The superstates can be obtained easily by

traversing G in a *depth-first search* manner while accumulating operations. A new superstate is built when we encounter a `wait` operation. Then function COMPUTE-TRACES computes all traces in ss and S. Note that we temporarily break loop back-edges, therefore there will be finite number of traces for ss and S. For each trace $\pi \in \mathcal{T}_1$, functions EXTRACT-TRACE-COND and FIND-TRACE will pair up traces in ss and S according to the trace condition C. We `assert` that each paired traces should be executed in the same number of cycles, and within each cycle, function HAS-1-1-MAPPING checks that I/O operations have one-to-one mappings between each superstate s_i^s and operation segment s_i^g. If there are no assertion failures after checking all traces, Algorithm 2 returns true, meaning that STP S is a valid Cycle-Fixed scheduling of CDFG G.

Algorithm 3: CHECK-SUPERSTATE-FIXED-MODE(G, S)

1 $ss \leftarrow$ BUILD-SUPERSTATES(G)
2 $\mathcal{T}_1 \leftarrow$ COMPUTE-TRACES(ss)
3 $\mathcal{T}_2 \leftarrow$ COMPUTE-TRACES(S)
4 **foreach** $\pi = [s_1^s, s_2^s, \ldots, s_m^s] \in \mathcal{T}_1$ **do**
5     $C \leftarrow$ EXTRACT-TRACE-COND(π)
6     $\pi' = [s_1^g, s_2^g, \ldots, s_n^g] \leftarrow$ FIND-TRACE($\mathcal{T}_2, C$)
7     $start \leftarrow 0$
8   **for** $i \leftarrow 0$ **to** m **do**
9         $end \leftarrow$ FIND-SHORTEST-SEGMENTS($\pi', start, s_i^s$)
10        $seg \leftarrow [s_{start}^g, \ldots, s_{end}^g]$
11   **assert** HAS-1-1-MAPPING($\mathcal{F}_{rw}(s_i^s),\ \mathcal{F}_{rw}(seg)$)
12   **assert** HAS-1-1-MAPPING($\mathcal{F}_{w}(s_i^s),\ \mathcal{F}_{w}(s_{end}^g)$)
13        $start \leftarrow end + 1$
14 **return true**

Algorithm 3 checks if STP S is a valid Superstate-Fixed scheduling of CDFG G. Similarly, the algorithm computes the traces of G and S. Each superstate may be "stretched" into multiple state after scheduling. For each trace pair $\pi \in \mathcal{T}_1$ and $\pi' \in \mathcal{T}_2$, we use function FIND-SHORTEST-SEGMENTS to find the corresponding trace segments that were "stretched" from a particular superstate. Function FIND-SHORTEST-SEGMENTS(π', $start$, s_i^s) finds the trace segments $seg = s^g_{start}, \ldots, s^g_{end}$ with minimum length which starts from *start* and ends at *end*, such that $\mathcal{F}_{io}(s_i^s) \subseteq \mathcal{F}_{io}(seg)$. We then `assert` that I/O operations have one-to-one mappings between each s_i^s and trace segments *seg*, which means that all I/O operations are within the bound of the superstate. We finally `assert` that `write` operations have one-to-one mappings between s_i^s and trace segment s^g_{end} (the last segment in *seg*), which means that all *write* operations within a superstate are scheduled to the last "stretches" state.

4.4 EXPERIMENTAL RESULTS

Table 4.1: Summary of Evaluation on S2CBench Benchmark

App. Domain	Design	Lines of C	Lines of RTL	# Processes.	# Functions.	Time (s)
Security	`AES CIPHER`	429	3941	1	11	8.89
	`KASUMI`	415	3602	2	5	0.44
	`MD5C`	467	4105	1	7	9.72
	`SONW 3G`	522	3121	1	11	1.54
Media Proc.	`QSORT`	204	865	1	1	0.07
	`SOBEL`	269	1191	1	2	0.15
	`ADPCM`	270	370	1	3	0.05
	`FIR`	176	561	1	2	0.07
	`DECIMATION`	422	3267	1	1	9.14
	`INTERPOLATION`	231	1721	1	1	0.18
	`IDCT`	450	4266	2	2	1.08
	`DISPARITY`	634	4355	4	4	9.06

We have implemented our scheduling validation algorithms in OCaml. We

```
1 void block::thread() {
2   int accu = 0;
3   wait();
4   while(1) {
5     Out.write(a);
6     Out.write(a+1);
7     wait();
8   }
9 }
```

(a)

```
1 void block::thread() {
2   int accu = 0;
3   wait();
4   while(1) {
5     Out.write(a);
6     add_state(); // an extra cy
  cle is added
7     Out.write(a+1);
8     wait();
9   }
10 }
```

(b)

Figure 4.4: An example of incorrect scheduling of signal I/O. (a). Design before scheduling, where a signal output `Out` is written twice with different values, however, only the last write is visible and valid. (b). The design after scheduling, where two writes of `Out` are scheduled to two different cycles. For simplicity, we use function `add_state()` to represent the scheduling transformation will add a new state on that line.

applied scheduling validation algorithms to designs from S2CBench [66], which is a publicly available behavioral synthesis benchmark suite containing 13 ESL designs written in synthesizable SystemC. The designs were synthesized by a commercial synthesis tool. We conducted our experiments on a workstation with Debian 7.1 running on a 2.93 GHz Intel Xeon X3470 processor with 8 GB of memory. Table 4.1 shows the statistics of the experiments. We can validate each of the 12 designs within 10 seconds. In the benchmark, the `FFT` design is not shown, because the floating point data type in it is not accepted by the synthesis tool.

```
1  void block::thread2() {
2    var = 0; // var is a variable
3    Out.write(0); // output
4    sig.write(0); // sig is a signal
5    wait();
6    while(1) {
7      var ++;
8      sig.write(var);
9      sc_uint<16> j = sig.read();
10     Out.write(j);
11     wait();
12   }
13 }
```

(a)

```
1  void block::thread2() {
2    var = 0; // var is a variable
3    Out.write(0); // output
4    sig.write(0); // sig is a signal
5    wait();
6    while(1) {
7      var ++;
8      sig.write(var);
9      add_state(); // an extra cycle
   is added
10     sc_uint<16> j = sig.read();
11     Out.write(j);
12     wait();
13   }
14 }
```

(b)

Figure 4.5: An example of incorrect scheduling of signal I/O. (a). Design before scheduling, where a local signal `sig` is written and then read at the same cycle. The read statement takes the old value. (b). The design after scheduling, where the write and read statements are scheduled to two different cycles, then the read takes the new value. For simplicity, we use function `add_state()` to represent the scheduling transformation will add a new state on that line.

SystemC allows users to model time and concurrency. In particular, according to the SystemC standard [6], if a given signal is written multiple times within an evaluation phase, the last write wins. That means all writes other than the last one is invisible in the simulation. The scheduling transformation must preserve signal behavior during scheduling. We found two bugs in the synthesis tool which violate the specification of SystemC. For simplicity, we provide the simplified version of the programs, showing both pre- and post-scheduling designs in SystemC for easy understanding.

Fig. 4.4(a) shows a design before scheduling, where `Out` is an output signal of type `sc_uint<16>`. According to the specification, the write statement `Out.write(a)` in line 5 is invisible and invalid, therefore should be eliminated. In another word, the only observable behavior should be the write statement in line 6. However, as shown in Fig. 4.4(b), the scheduling transformation scheduled the two writes into two different states. In this case, both of the two writes are observable, which violates the SystemC standard. Algorithm 3 detects this violation by checking that the two `write` statements are scheduled to two different states.

Fig. 4.5 shows another scheduling bug. According to the SystemC standard, If a signal is written and read during the same evaluation phase, the old value will be read. The value written will be available in the subsequent evaluation phase. In Fig. 4.5(a), signal `sig` is written and read in the same cycle. Therefore variable `j` will take the old value of `sig`. However, after scheduling, as shown in Fig. 4.5(b), the read of `sig` is scheduled to the next cycle after the write. Variable `j` will take the new value of `sig` instead of the old one.

4.5 RELATED WORK

Anderson [11] reports an early effort on the verification of *as soon as possible* scheduling transformation using theorem proving. Narasimhan *et al.* [55] used theorem proving approach to verification of force-directed list scheduling algorithm for resource-constrained scheduling in high-level synthesis. Karfa *et al.* [42] develops techniques for more automated equivalence checking on scheduling transformation. This framework converts the designs before and after scheduling transformation into Finite State Machine with Datapath (FSMD) models, then checks the equivalence of two FSMD models. The major difference between the above approaches and our research is the observation that scheduling transformations can be extricated from compiler transformations and handled as a verification of partitioning.

This permits efficient static checking to validate these transformations, obviating expensive theorem proving or symbolic simulation techniques used in previous work. The efficiency is critical in enabling application of our approach on large-scale designs.

4.6 SUMMARY

Scheduling transformation is an important synthesis phase that significantly affects the quality of the synthesis results. Therefore scheduling transformation usually uses complex algorithms to satisfy user-specified timing and resource constraints. In this chapter, we propose a simple and efficient approach to validating scheduling transformations in behavioral synthesis. We characterize different widely used scheduling modes, formalize equivalence relations to compare designs scheduled by different modes, and propose efficient algorithms to validate designs scheduled by each mode. Experiments on 12 synthesizable designs in S2CBench show that our approach can successfully validate all designs within a few seconds. Furthermore, our approach detected bugs in a commercial behavioral synthesis tool.

Chapter 5

SCALING BACK-END RTL GENERATION CHECKING

5.1 EQUIVALENCE CHECKING FRAMEWORK

Previous work [36, 64] developed a sequential equivalence checking (SEC) framework for behavioral synthesis. Fig. 5.1 shows the framework. It uses a formal structure, *Clocked Control/Data Flow Graph* (CCDFG), as a uniform design abstraction after scheduling transformation, and takes the generated RTL circuit, together with equivalence mapping points obtained from the behavioral synthesis tool.

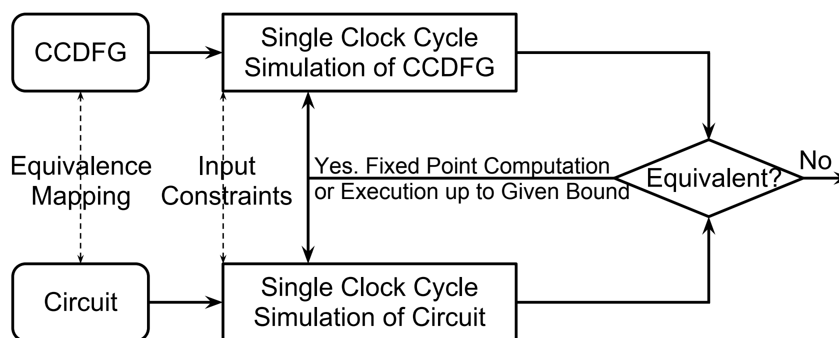


Figure 5.1: Dual-rail cycle-based symbolic simulation of a CCDFG and RTL circuit.

The key ingredients of the framework were (1) the use of a formal structure, CCDFG as a uniform design abstraction, (2) a certified sequence of high-level transformations to reduce the abstraction gap, (3) an SEC algorithm based on dual-rail symbolic simulation between CCDFG and RTL, and (4) optimizations that

enable compositional application of SEC exploiting internal cutpoints and modular structures. Experimental results reported successful certification of synthesized designs with tens of thousands of lines of RTL for ESL specifications of a number of cryptographic algorithms.

Unfortunately, the above approach cannot directly handle certification of designs from other domains that involve considerably less structure. In particular, one key requirement to achieve compositionality in SEC is the availability of equivalent internal operations or modules between the abstract CCDFG and the corresponding RTL, which are then used as *cutpoints*. However, we found that for many synthesized ESL designs, there are very few internal operations that preserve such equivalence in the presence of design and implementation optimizations, thus undermining compositionality and hence scalability. We present techniques for SEC between ESL designs and synthesized RTL, in the presence of optimizations that violate local equivalences of internal signals. Our key observation is that there are two key sources of local inequivalence between CCDFG and RTL:

- **Operation Gating:** Behavioral synthesis tools often optimize the RTL by introducing control structures or “guards” to ensure that certain operations are executed only when their results are relevant to downstream computation, and turned off otherwise. Such gated operations are functionally equivalent to the behavioral specification only under these guards. This makes such an operation difficult to identify; more problematically, it precludes the naive approach of using it as a cutpoint by verifying it in isolation and replacing it with an uninterpreted function in the CCDFG and RTL.

- **Global Variables:** Global variables are used commonly in ESL as a *design optimization*: the user can then define some design functionalities as implicit side effects of other design modules, reducing the lines-of-code in ESL description and thus improving compactness. Unfortunately, global variables

break the compositional approach of verifying modules compositionally, since the side effects on these variables must be accounted for during SEC.

In the next two sections, we present algorithms to enable compositional SEC for behavioral synthesis in the presence of the above design and implementation optimizations.

5.2 HANDLING OPERATION GATING OPTIMIZATION

The idea of operation gating is to add controlling predicates so that an operation is not executed when the value computed is irrelevant to downstream computation. Behavioral synthesis tools generate optimized RTL with operation gating to facilitate power-friendly hardware systems [21]. The transformation itself is complex, and its details are not germane to this paper. The characteristic of operation gating that is relevant to equivalence checking is that some operations have explicitly generated gating predicates in the synthesized RTL, when no such predicate appears in the CCDFG. The effects of the operation on the CCDFG and the RTL are then equivalent only when the gating predicate holds.

Consider synthesizing the code fragment shown in Fig. 5.2(a). According to the semantics of C, the multiplication operation in Line 3 (and the assignment of the result to c) must be executed regardless of the value of b. However, the result of multiplication is only relevant to the eventual return value f when the value of b is 1. In the RTL shown in Fig. 5.2(c), the multiplication operation is therefore gated by condition b' so it is only executed when b' has the value 1.

Unfortunately, operation gating breaks compositionality. Recall from Section 5.1 that a key optimization involved in scaling up SEC for behavioral synthesis is the utilization of *cutpoints*. Cutpoints entail pre-verification of equivalence between corresponding internal variables in the CCDFG and the RTL, which are then replaced by (equivalent) symbolic variables. However, since the output of a

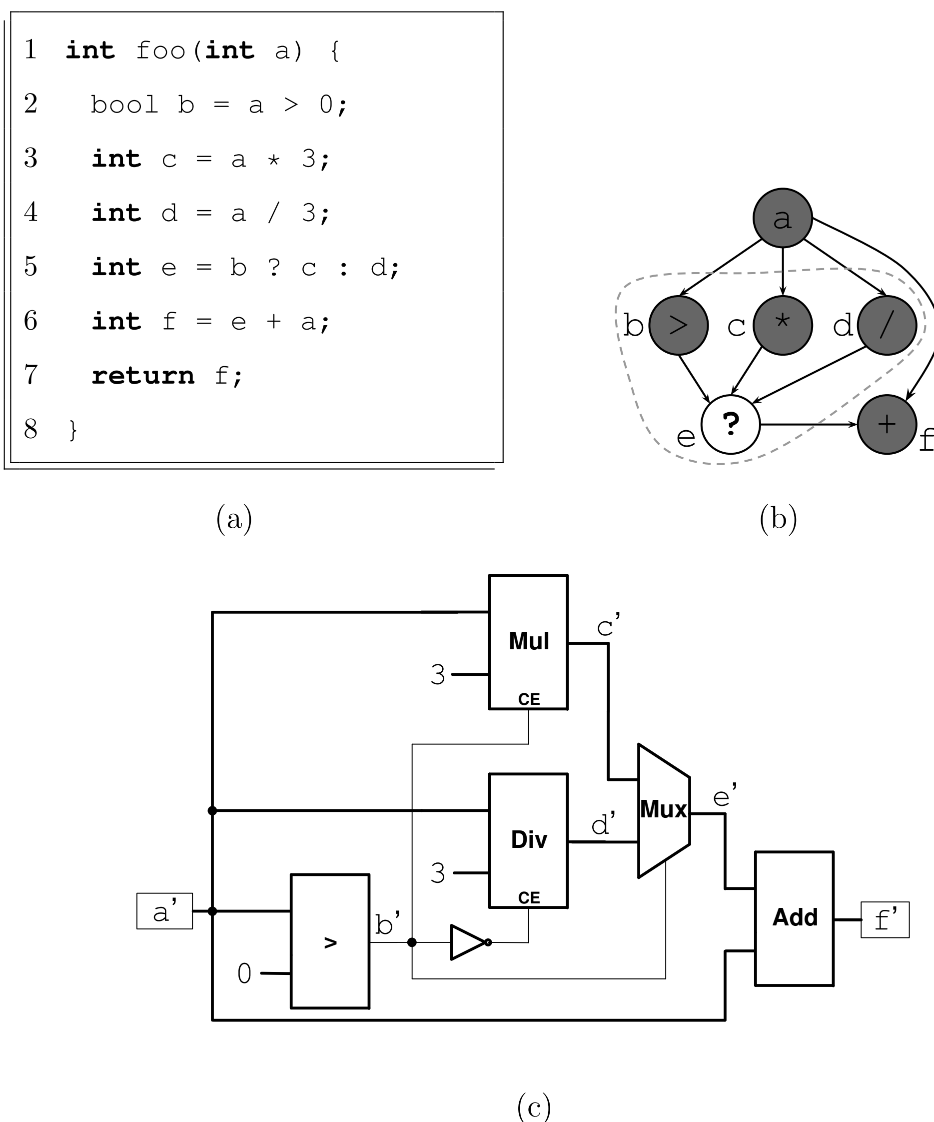


Figure 5.2: Operation gating example. (a) C code. (b) Data flow graph. (c) Schematic of generated RTL

gated operation is only equivalent when the gating condition is satisfied, its use as a cutpoint will cause the pre-verification to report inequivalence, breaking the compositional SEC flow.

To address this issue, we develop a *relaxed checking* algorithm for compositional SEC between a CCDFG G and a circuit M that tolerates local, "irrelevant" inequivalences for individual variables. The key idea is to continue dual-rail symbolic

simulation even when a local inequivalence is encountered, but keep track of these inequivalences so that we can check if they are irrelevant during subsequent symbolic simulation. Algorithm 4 provides a high-level presentation of our approach. Here t_k denotes the scheduling step in clock cycle k, $EMap$ maps an operation op in CCDFG to combinational node in M, and x_k, s_k, i_k denote CCDFG state, circuit state, and inputs in clock cycle k respectively. At any point, the algorithm maintains a set, called $InEqSet$, of currently encountered variable inequivalences. For our example, in Fig. 5.2, $InEqSet$ will record the inequivalent pairs $\langle c, c' \rangle$ and $\langle d, d' \rangle$ between the CCDFG and the RTL when simulating Lines 3 and 4 respectively. During subsequent symbolic simulation, whenever an equivalence is discovered between variables in G and M, we check if that makes any of the inequivalences currently in $InEqSet$ irrelevant. For instance, when simulating Line 5 we find that e and e' are equivalent irrespective of the inequivalences between $\langle c, c' \rangle$ and $\langle d, d' \rangle$, making these two inequivalences irrelevant. When symbolic simulation terminates, one of two outcomes is possible.

- $InEqSet$ is empty, meaning all inequivalences encountered have been resolved (*i.e.*, found irrelevant). The algorithm then reports G and M to be equivalent.

- $InEqSet$ still contains some inequivalences. This means that some operations found inequivalent during symbolic simulation remain relevant even after fixpoint is reached. Thus the algorithm returns G and M to be inequivalent (and outputs the unresolved inequivalences).

Algorithm 4 makes use of two key subroutines, FIND-GATING-INFO and RESOLVE-INEQ to do the analysis of irrelevance of local inequivalences. To describe these subroutines we first need a key definition below. For this definition, recall that a Data Flow Graph (DFG) is a directed graph $G_D = (V, E)$, where each $v \in V$ is a variable in the program, each edge $(x, y) \in E$ represents a data dependency,

Algorithm 4: RELAXED-CHECKING(G, M)

```
1  k ← 0                                                  ▷ Set clock cycle to 0
2  InEqSet ← ∅                                            ▷ Empty inequivalence set
3  G_Info ← FIND-GATING-INFO(G)
4  while not (checking bound or fix-point reached) do
5      x_{k+1} ← SIM-CCDFG(G, t_k, x_k, i_k)
6      s_{k+1} ← SIM-RTL(M, s_k, EMap(i_k))
7      foreach op_g ∈ t_k do
8          op_m ← EMap(op_g)                              ▷ find the op in circuit M
9          if not IS-EQUAL(op_g, op_m) then               ▷ SMT query
10             InEqSet ← InEqSet ∪ {⟨op_g, op_m⟩}
11         else
12             RESOLVE-INEQ(InEqSet, G_Info, op_g, op_m)
13     k ← k + 1
14 if |InEqSet| = 0 then                                  ▷ All inequivalences resolved
15     return true
16 else
17     print InEqSet                                      ▷ Report all inequivalences
18     return false
```

meaning the value of variable y depends on the value of variable x. Furthermore, we will assume that each node in G_D is labeled with an operation (*e.g.*, `add`, `mul`, etc.).[1]

Definition 5.1 (Post Dominance)**.** Let G_D be a Data Flow Graph for a design, and u and v be two variables. We say that u is *post-dominated* by v in G_D iff $u \neq v$

[1]This assumption is valid in our case since the instructions in a CCDFG are in static single assignment (SSA) form; thus each variable can be uniquely associated with one operation.

and any path that starts from u goes through v.

Remark 5.1. *Post-dominance is a common concept in compiler literature [27], although it is typically defined with respect to the Control Flow Graph instead of the DFG as above. The definition extends to a CCDFG G by taking G_D to be the DFG component of G. Given a variable mapping $EMap$, we can also extend the notion to the circuit M: a variable u' in the circuit is* post-dominated *by v' if and only if (1) there are variables u and v in G that are mapped to u' and v' respectively, and (2) u is* post-dominated *by v. Thus we will often call $\langle u, u' \rangle$ to be* post-dominated *by $\langle v, v' \rangle$.*

The definition of post dominance guarantees that every path from u in G_D must go through v, *e.g.*, in the example in Fig. 5.2(b), the variables c and d are post-dominated by e. Let $\langle u, u' \rangle$ be post-dominated by variables $\langle v, v' \rangle$ in G and M respectively. Then if v and v' are equivalent, it follows that from the perspective of any pair of corresponding variables $\langle x, x' \rangle$ that are descendants of $\langle v, v' \rangle$, the equivalence or inequivalence of $\langle u, u' \rangle$ does not matter. For instance, in Fig. 5.2, if e and e' are equivalent, then the inequivalence of c and c' is irrelevant. This observation leads to the theorem below that is an easy consequence of data flow.

Theorem 5.1. *Suppose G is a CCDFG and M is a circuit such that the following hold: (1) variables $\langle v, v' \rangle$ are equivalent in G and M, and (2) $\langle u, u' \rangle$ are post-dominated by $\langle v, v' \rangle$ respectively. Let $\langle x, x' \rangle$ be arbitrary corresponding descendants of $\langle v, v' \rangle$. Then the equivalence between u and u' is irrelevant to the equivalence of x and x'.*

We now discuss the two subroutines.

FIND-GATING-INFO. This subroutine finds the potential gating information for a CCDFG G. A *potential gating information* is a list of pairs $\langle v, U \rangle$ where v is a variable and U is a set of variables such that each variable $u \in U$ is post-dominated

by v. Theorem 5.1 guarantees that if v is equivalent to v' in G and M then the inequivalences of variables in U are irrelevant. Our implementation exploits the underlying LLVM constructs and information from the synthesis to efficiently determine relevant post dominance information. In particular, LLVM has a special `select` instruction of the form `y = select cond x1 x2`; the synthesis tool typically targets the condition variable of `select` instructions for operation gating.[2] Function FIND-GATING-INFO crawls over the data flow graph of CCDFG G, first identifying each `select` instruction; for each `y` it then finds all variables that are post-dominated by `y` recursively.

RESOLVE-INEQ. This function tries to resolve inequivalences in $InEqSet$ using the gating information found by FIND-GATING-INFO. Let $\langle v, v' \rangle$ be determined to be equivalent during symbolic simulation. Then we find the set U such that $\langle v, U \rangle$ is a pair computed by FIND-GATING-INFO. From the above discussion, inequivalences involving variables in U are irrelevant, therefore dropped from $InEqSet$.

5.3 HANDLING GLOBAL VARIABLES

Modular design provides several advantages by breaking the design into modules. One key optimization presented in previous work [36] is modular analysis. The basic idea is to check each module individually in a bottom up manner.

- For each module M, check the equivalence of CCDFG and RTL.
- When checking module M' that calls M, replace the invocation of M in both CCDFG and RTL by equivalent uninterpreted functions.

However, global variable usages break this modular view, and one must account for side effects on these variables while performing modular analysis. Note that

[2] U need not be the *complete* set of variables post-dominated by v. This permits us to merely consider conditions in the LLVM `select` instruction as potential gating information. This runs the risk of possible spurious SEC failures. However, in our experience, this check has been sufficient.

Algorithm 5: GET-EXTENDED-SIGNATURE(f)

```
1  I ← PARAMETERS(f)
2  O ← OUTPUTS(f)
3  V_G ← FIND-ALL-GLOBALS(f)
4  foreach v ∈ V_G do
5      switch USAGE-TYPE(v) do
6          case R :  I ← I ∪ {v}                    ▷ read-only
7          case W :  O ← O ∪ {v}                    ▷ write-only
8          case RW :                                ▷ read-and-write
9              I ← I ∪ {v}
10             O ← O ∪ {v};
11 return ⟨I, O⟩
```

while the side effects are implicit for high-level design descriptions (and hence CCDFGs), they are explicit on the synthesized RTL since the synthesis tool usually places the global variable on the interface when generating RTL.

We employ the similar approach to handling global variables as proposed in Section 3.3. Different from the approach employed in front-end, where sub-functions are invoked in the form of function calls, sub-modules in the RTL are instantiated as sub-module. We compute an *extended signature* for a module that accounts for globals explicitly. Algorithm 5 shows how to compute the extended signature of a module. The key idea is to analyze the module to determine the globals used in the module. The parameters of the module are then extended to include *read-only* and *read-write* globals among the inputs and *write-only* and *read-write* globals among the outputs. Extended signatures explicitly account for global variables during modular analysis.

5.4 EXPERIMENTAL RESULTS

Table 5.1: Summary of CHStone Benchmark for Equivalence Checking of Back-end RTL Generation

App. Domain	Design	Lines of code		C Functions	RTL Modules
		C	RTL		
Arithmetic	DFADD	526	3722	17	5
	DFDIV	436	5192	19	4
	DFMUL	376	3115	16	2
	DFSIN	755	11224	31	8
Microprocessor	MIPS	232	2944	1	1
Media Processing	ADPCM	541	14935	15	5
	GSM	393	5598	12	4
	JPEG	1692	32846	30	17
	MOTION	583	6168	13	5
Security	AES	716	11869	11	7
	BLOWFISH	1406	17420	6	4
	SHA	1284	18819	8	4

5.4.1 Performance Evaluation

We have applied our framework to certify synthesized RTL for all the ESL designs in the CHStone benchmark. CHStone is a publicly available benchmark suite for behavioral synthesis, that includes twelve designs selected from different application domains. We used a commercial behavioral synthesis tool to synthesize the RTL. The most complex design in the benchmark is `JPEG` which has more than 32K lines of RTL code. For our experiments we have used the benchmark designs as is with one modification: two designs, `JPEG` and `MOTION`, used double pointers

Table 5.2: Summary of Evaluation on CHStone Benchmark for Equivalence Checking of Back-end RTL Generation

Design	Operation Gating	Global Variables[a]			Time (s)	Memory (MB)
		R	W	RW		
`DFADD`	Yes	4	0	1	174.9	169.34
`DFDIV`	Yes	4	0	1	6946.1	594.87
`DFMUL`	Yes	4	0	1	63.5	75.31
`DFSIN`	Yes	6	0	1	7151.3	603.50
`MIPS`	No	1	0	0	250.4	125.21
`ADPCM`	No	15	19	63	68.2	105.45
`GSM`	Yes	4	0	0	49.6	83.07
`JPEG`	Yes	30	14	17	2187.3	375.90
`MOTION`	Yes	9	0	4	1515.1	408.77
`AES`	Yes	4	0	5	170.7	106.59
`BLOWFISH`	No	3	0	4	44.9	91.89
`SHA`	No	3	0	4	6.0	89.04

[a] R means read-only, W means write-only, and RW means read-and-write.

to represent two-dimensional arrays; these were modified to eliminate the double-pointer and represent the arrays explicitly. The reason has to do with the quirks of the synthesis tool used in this experiment. The synthesis tool inlines functions that have double pointers, thus flattening the module structure in the synthesized RTL. In addition to generating significantly larger RTL, this also destroys the module structure in the synthesized design. Since scalability of *modular analysis* (in the presence of design optimizations) is the key target of the experiments, we found the original designs unsuitable as targets for evaluation. The experiments

were conducted on a workstation with 3GHz Intel Xeon processor and 8GB memory. For each design, we checked the equivalence between its CCDFG and RTL via dual-rail symbolic simulation, which symbolically simulates the CCDFG and RTL clock cycle by clock cycle. After each clock cycle, we checked the equality of mapped variables in the CCDFG and RTL by the MathSAT SMT solver [13]. We also applied cutpoints, cut-loop, and modular analysis optimizations when checking each design.

Table 5.1 and 5.2 show the results of the experiments. The `JPEG` design takes about 36 minutes with 375.9 MB memory usage. The maximum certification time is required for `DFSIN`, which takes around 119 minutes with 603.5 MB memory usage. The experiment results demonstrate that independent of application domain our framework scales up to designs of practical complexity. No other SEC framework to our knowledge can handle behaviorally synthesized designs at this scale. Furthermore, only `MIPS` can be certified without handling operation gating and global variable optimizations.

5.4.2 A Behavioral Synthesis Bug

Our experiments found a bug in the synthesis tool during the certification of the `MOTION` design, which is a C implementation of a motion vector decoding algorithm for MPEG-2. Fig. 5.3 shows the source code fragment that triggers the bug. Here `ld_Bfr` is a global variable. In function `Get_Bits`, the return value `Val` is computed by right-shifting `ld_Bfr`. After `Val` is computed, `ld_Bfr` is updated in the subroutine `Flush_Buffer`. The update performed by `Flush_Buffer` does not affect the return value. Fig. 5.4(a) shows the RTL implementation synthesized by the behavioral synthesis tool. The global variable `ld_Bfr` is synthesized to a register outside of module `Get_Bits`. The output of `Get_Bits` is thus a combinational circuit with `ld_Bfr` as input. Therefore, when sub-module `Flush_Buffer` produces a new data for `ld_Bfr`, the new data is propagated to the output in the same clock

```
1  void Flush_Buffer(int N) {
2    // modify the global variable
3    ld_Bfr = update(N, ld_Bfr);
4  }
5  unsigned int Get_Bits(int N){
6    unsigned int Val;
7    Val = ld_Bfr >> (32 - N);
8    Flush_Buffer(N);
9    return Val;
10 }
```

Figure 5.3: Simplified C source code of the `MOTION` example.

cycle, leading to a wrong output.

The bug is caused because behavioral synthesis applies aggressive transformations to minimize resource usage. As can be seen by comparing Figs. 5.4(a) and 5.4(b), the synthesis tool in this case eliminates a register without correctly taking into account the side effect on the global variable. Such subtleties reinforce the need for SEC for certification of synthesized RTL designs. The bug has been confirmed by developers of the synthesis tool and fixed in a new release.

5.5 RELATED WORK

Recently increasing sophistication of behavioral synthesis has resulted in several SEC optimizations to scale up certification of synthesized RTL [34, 36, 41, 45, 47]. For instance, Koebl *et al.* [44] provide a good overview of research in SEC between high-level and RTL designs. Vasudevan *et al.* [70] introduce *sequential compare points* as a set of observable signals to be compared between high-level designs and RTL. There are commercial tools [16, 45] that can apply SEC between RTL and high-level (C/C++/SystemC) models. However, we have found no published

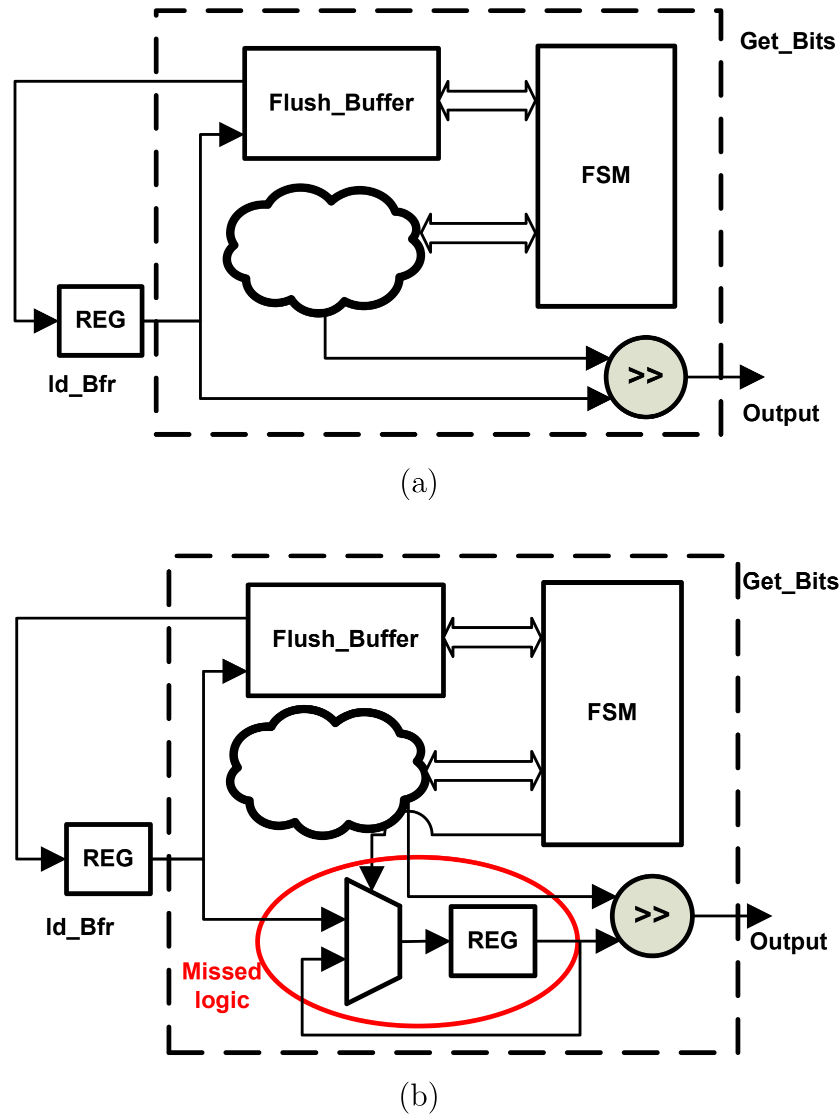


Figure 5.4: Bug found in the `MOTION` example, where an important register is eliminated. (a). Wrong RTL (b). Correct RTL

results on approaches to handling design and implementation optimizations in any certification framework for behavioral synthesis. There has however been research on handling such optimizations in SEC comparing RTL and netlist designs. Baumgartner *et al.* [12] discuss an approach for invariant generation to address the conditional equivalence checking problem for optimizations including clock gating

and power gating. Moon *et al.* [54] propose equivalence checking techniques that exploit well-partitioned circuit structures.

5.6 SUMMARY

In this chapter, we presented algorithms that enables compositional SEC for behavioral synthesis in the presence of the two design and implementation optimizations.

1. We develop an algorithm for *relaxed* SEC that includes identification and compositional use of gated variables. The approach tolerates local, "irrelevant" inequivalences between gated variables and their RTL counterparts, as long as the inequivalences are resolved during symbolic simulation of downstream computation.

2. We develop an approach to modeling the side effects of global variables explicitly and show how the approach can then be used with modular analysis.

The algorithms, albeit not individually complex, have been carefully developed to (1) exploit the constraints and invariants available from the behavioral synthesis process, and (2) reinforce the available SEC optimizations, facilitating smooth integration. As a result, our back-end SEC scales to practical designs: it can handle all designs of the CHStone benchmark, some of which have more than 32K LoC synthesized RTL. We do not know of any other tool that can handle diverse designs at this scale, and the algorithms presented here are crucial to this scalability. Finally, we found a subtle bug in an optimization of the behavioral synthesis tool itself, demonstrating both the need for certification of behaviorally synthesized designs and the importance of SEC in general and our framework in particular to achieve such certification.

Chapter 6

INTERFACE SYNTHESIS CHECKING

Previous chapters mainly focus on module level (block level) equivalence checking in behavioral synthesis. We reason about the equivalence between the high-level design and the synthesized low-level implementation on a module-by-module basis. In this chapter, we focus on the interfaces between modules in a design. In particular, we focus on detecting deadlocks introduced by interface synthesis.

Most behavioral synthesis tools support interface synthesis, which allows the user to map the interfaces of a high-level design to some pre-defined interface components [28, 33, 73], *e.g.*, the user can map an array on the argument list of a high-level function to a memory interface in the RTL. Interface synthesis may add additional RTL ports with associated I/O handshaking protocols. Some interface components have complicated timing and communication protocols. Interface synthesis is also a complex and error-prone process. In practice, bugs appear often on the interfaces between different modules in a design.

In the high-level sequential programs in C, functions communicate with each other through arguments passing and global variables sharing, and functions are executed one at a time. However, in the synthesize hardware, there can be concurrent blocks that are connected by synthesized interfaces of different types. Converting a high-level sequential specification into concurrent RTL blocks that are connected with interfaces of different handshaking protocols may introduce deadlocks. In this chapter, we first introduce different commonly used interfaces and their handshaking protocols. We then discuss the deadlock situations, and present an approach to detection of deadlocks.

6.1 INTERFACE SYNTHESIS

For C based designs, a function communicates with the other part of the system through global variables, function arguments, and function return value. Interface synthesis transforms each individual function argument and the return value into RTL interfaces with user-specified communication protocols. Fig. 6.1 shows an example of interface synthesis of a C function into an RTL block. We can see that the input and outputs are synthesized into interfaces with different handshaking protocols.

Most high-level synthesis tools support different handshaking protocols: *e.g.* no handshaking ($\mathcal{P}_{\text{NONE}}$), one-way handshaking ($\mathcal{P}_{\text{ACK}}$ and $\mathcal{P}_{\text{VLD}}$), two-way handshaking ($\mathcal{P}_{\text{ACK-VLD}}$), streaming ($\mathcal{P}_{\text{FIFO}}$) and memory ($\mathcal{P}_{\text{MEM}}$) interfaces:

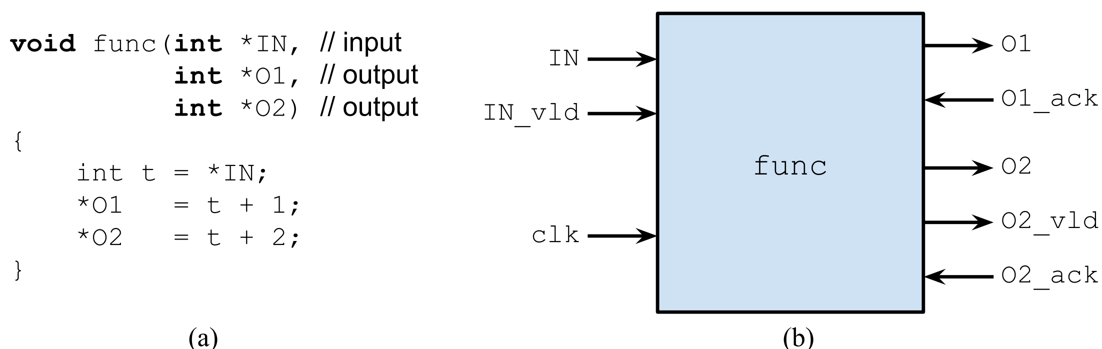


Figure 6.1: Example of interface synthesis. (a). A simple C function with one input and two outputs. (b). Synthesized block diagram of the C function. Input `IN` has an associated valid signal `IN_vld` to indicate when `IN` is ready to be read. Output `O1` has an associated acknowledge signal `O1_ack` to allow the downstream block to acknowledge block 'func' that the output data `O1` has been read. Output `O_2` has both valid and acknowledge signals.

- $\mathcal{P}_{\text{NONE}}$: wire or register interface will be created. There is no additional signal generated to indicate when data is read or written.

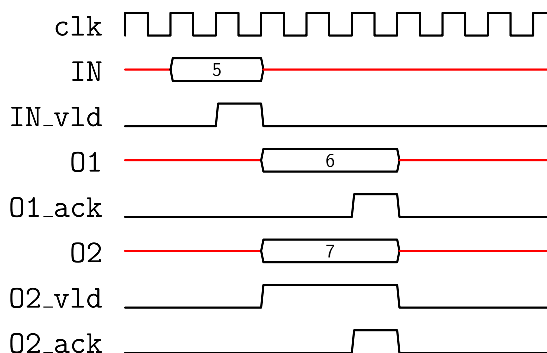


Figure 6.2: Timing diagram of the synthesized block 'func'.

- $\mathcal{P}_{\text{ACK}}$: for an input port, an additional output port is created to acknowledge the upstream block that the data has been read. Similarly, for an output port, an additional input port will be created to acknowledge the downstream block that the data has been read. The output will be held until the acknowledge signal is received. For example, the output `O1` of the C function '`func`' in Fig. 6.1(a) is synthesized with the protocol $\mathcal{P}_{\text{ACK}}$. An additional input port `O1_ack`, as shown in Fig. 6.1(b), is generated. We can see from the timing diagram shown in Fig. 6.2 that the value 6 is held until signal `O1_ack` goes to high. It is worth noting that when using $\mathcal{P}_{\text{ACK}}$ protocol, a block cannot write any data until the previous data is received by the downstream block.

- $\mathcal{P}_{\text{VLD}}$: an additional port will be created to indicate that the data is ready to be read. When a block tries to read a value from an upstream block, it waits until the associated valid signal goes to high. For example, the input `IN` of the C function '`func`' in Fig. 6.1(a) is synthesized with the protocol $\mathcal{P}_{\text{VLD}}$. An additional input port `IN_vld`, as shown in Fig. 6.1(b), is generated. In the timing diagram shown in Fig. 6.2, we can see that signal `IN_vld` goes to high at the third clock cycle, which indicates that the value 5 on the port `IN` is valid to be read.

- $\mathcal{P}_{\text{ACK-VLD}}$: this two-way handshaking protocol combines the protocol $\mathcal{P}_{\text{ACK}}$ and $\mathcal{P}_{\text{VLD}}$, which not only indicates when the data is valid to be read, but also allows the downstream block to acknowledge the data has been read. For example, the output `O2` in Fig. 6.1 is synthesized with $\mathcal{P}_{\text{ACK-VLD}}$ protocol. From the timing diagram in Fig. 6.2, we can see that the value 7 stays valid and held (for three cycles in this example) by block `func` until block `func` receives the acknowledge signal from the downstream block.

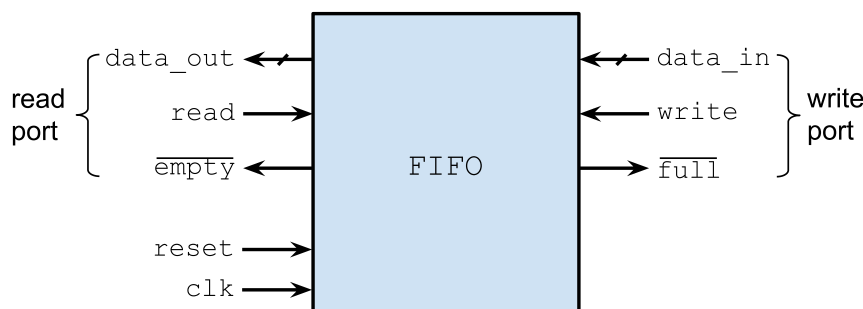


Figure 6.3: Block diagram of a FIFO with read and write ports. Signals $\overline{\texttt{empty}}$ and $\overline{\texttt{full}}$ indicate the emptiness and fullness of the FIFO.

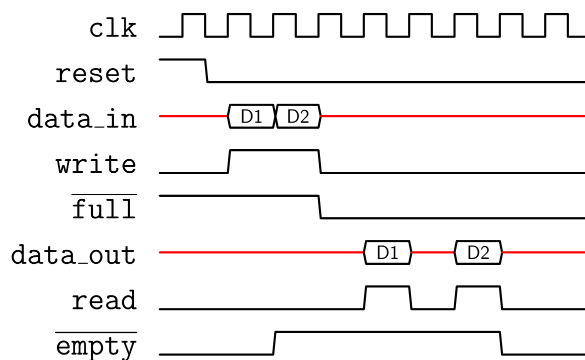


Figure 6.4: Timing diagram of a FIFO with depth of 2.

- $\mathcal{P}_{\text{FIFO}}$: this interface protocol is often used for streaming data between two blocks. Fig. 6.3 shows the block diagram of a FIFO (first-in, first-out) with a read and a write port. Each FIFO has a user-specified depth. Signal $\overline{\texttt{empty}}$

goes to low if the FIFO is empty. Similarly, signal $\overline{\texttt{full}}$ goes to low if the FIFO is full. Fig. 6.4 shows the timing diagram of a FIFO with depth of 2. Suppose the FIFO is empty initially, then after two writes to the FIFO, the FIFO is full (signal $\overline{\texttt{full}}$ goes to low). It becomes empty again after two reads. No reads can be performed when FIFO is empty, and no writes can be performed when the FIFO is full. In other words, reading of an empty FIFO or writing to a full FIFO is a blocking operation.

- $\mathcal{P}_{\text{MEM}}$: this interface protocol, which is often used for array arguments in C, is synthesized to connect to memories (ROMs and RAMs) in RTL implementations.

6.2 DEADLOCK DETECTION

6.2.1 Deadlock Example

C/C++ based untimed designs are required to be single-threaded programs by most behavioral synthesis tools . In reality, hardware blocks synthesized from functions may run concurrently. Designers can synthesize concurrent hardware blocks by applying constraints to single-threaded C/C++ functions or interfaces to improve throughput. However, making blocks run concurrently may introduce deadlocks. Among all available interfaces, FIFOs are often used to implement streaming behavior between blocks. Since FIFO read and write may be blocking operations, insufficient depth of FIFOs may lead to system deadlock. Most practical designs require designers to manually set the depth of each FIFO [28].

Consider an example shown in Fig. 6.5 and Fig. 6.6. Fig. 6.6 shows the synthesized design with three blocks: `BLOCK1`, `BLOCK2` and `BLOCK3` from the high-level specification shown in Fig. 6.5. Interfaces `B`, `C` and `D` are FIFOs of depth of 1, 2 and 1, respectively. Suppose that `BLOCK1` has a fixed latency of 1. It tries to push a new value to FIFO `B` and a new value to FIFO `C` at every clock cycle. Suppose

```
1   void BLOCK1(int *I, int *O1, int *O2) {
2     int t = *I;
3     *O1 = t + 1;
4     *O2 = t - 1;
5   }
6   void BLOCK2(volatile int *I, int *O) {
7     int t = *I;
8     t += *I;
9     t += *I;
10    *O = t/3;
11  }
12  void BLOCK3(int *I1, int *I2, int *O) {
13    *O = *I1 + *I2;
14  }
15  void top(int A, int E) {
16    int B, C, D;
17    BLOCK1(&A, &B, &C);
18    BLOCK2(&B, &D);
19    BLOCK3(&D, &E);
20  }
```

Figure 6.5: A design in C with three blocks.

`BLOCK2` has a latency of 3, it reads three values (one at each clock cycle), and then pushes the average of the three values into the FIFO `D`. `BLOCK3` reads the values from FIFO `C` and `D` at the same clock cycle, and returns the sum of the two values as output `E`. Suppose input `A` is available all the time. `BLOCK1` stalls after pushing two values to FIFO `C`, because FIFO `C` is full. `BLOCK2` now has already read two values from FIFO `B`, and is waiting for the third value in order to calculate the average. `BLOCK2` stalls because FIFO `B` is empty, since `BLOCK1` stalls and cannot

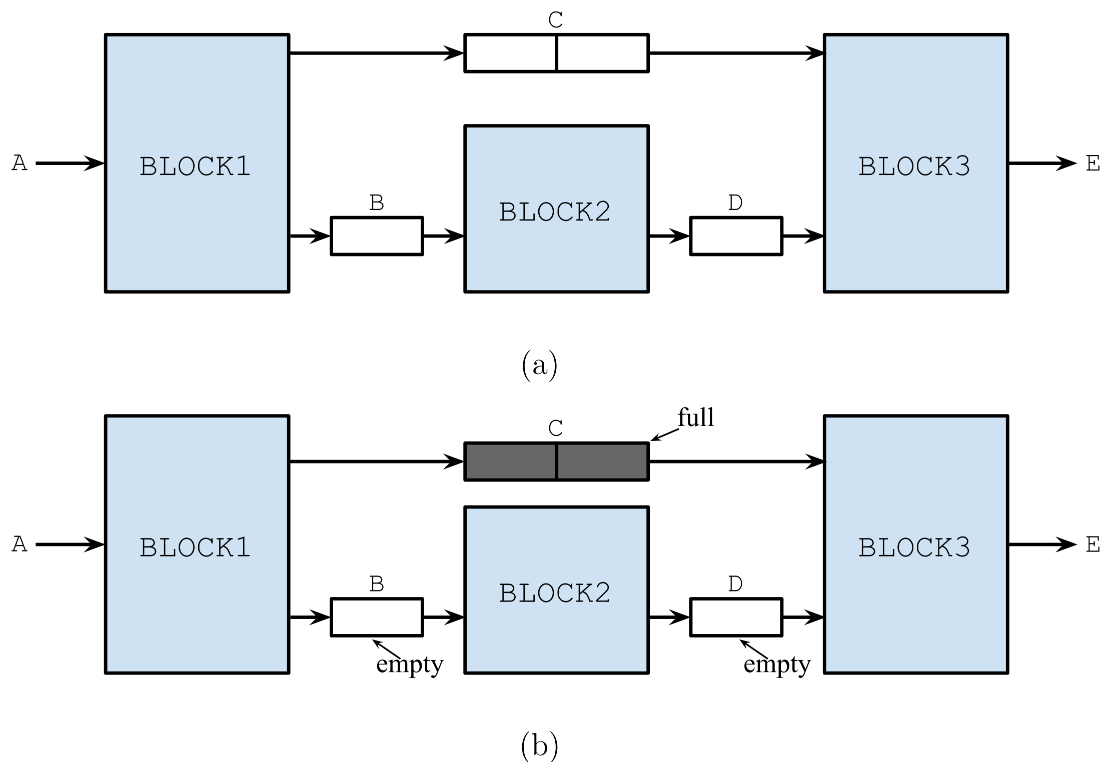


Figure 6.6: Deadlock example: a synthesized design with three blocks. Interface `C` is a FIFO of depth of 2. Interfaces `B` and `D` are FIFOs of depth of 1. (a). FIFOs are initialize to be empty. (b). Status of FIFOs when the design deadlocks.

produce more data to FIFO `B`. `BLOCK3` stalls because FIFO `D` is empty. As a result, the design deadlocks because every block is waiting for other blocks to make progress in order to continue the execution.

From the above example, we can see that synthesizing concurrently running blocks from a sequential specification could introduce deadlocks. It is a non-trivial task to analyze the possibilities of deadlock situations of a design when the design contains complex logic. Therefore, a deadlock detection approach is highly desired.

6.2.2 Deadlock Detection Algorithm

Assertion-based verification is a widely used verification methodology in hardware design community. Designers use assertion languages, such as Property Specification Language (PSL) [2] and SystemVerilog Assertions (SVA) [7], to capture the intent of a design, and use simulation or formal verification tools to verify that the design correctly implements that intent. In the context of deadlock detection, designers usually manually write the assertions in RTL to capture the deadlocks in a particular design in practice. However, synthesized RTL implementations from behavioral synthesis tools are generally not human readable. Systematically writing sufficient assertions to capture deadlocks and debugging deadlocks in behaviorally synthesized RTL designs are often difficult. Therefore, it is highly desired to design an algorithm to automatically generate the assertions for detection of deadlocks in the synthesized RTL designs. Since we can obtain the types of interfaces and the variables mappings between the CCDFG and the RTL design from the synthesis report, it is feasible to automatically generate the assertions to capture the deadlock conditions in the synthesized RTL designs.

Definition 6.1 (Blocking Interface)**.** An interface I of a module A is said to be a blocking interface of A if I/O operations in A of the interface I may need to wait until a certain condition is satisfied.

An I/O operation on a blocking interface is called a blocking I/O operation. For example, reading data from an interface with $\mathcal{P}_{\text{VLD}}$ protocol is a blocking I/O operation, because it may need to wait until the associated `valid` signal is high. Similarly, writing data to an interface with $\mathcal{P}_{\text{ACK}}$ protocol, reading or writing data from an interface with two-way handshaking protocol or $\mathcal{P}_{\text{FIFO}}$ are also blocking I/O operations.

Definition 6.2 (Interface Dependency)**.** Suppose that blocks A and B are connected with an interface I. We say A depends on B if there is blocking I/O

operation on interface I in A.

For example, if block A has an output with interface protocol $\mathcal{P}_{\text{VLD}}$, and block B takes A's output as input, then B depends on A. Because when B reads the input, it may need to wait until the data is available. However, A does not depend on B in this case, because there is no acknowledge signal from B, thus A can output at any time. If A and B are connected with an interface with $\mathcal{P}_{\text{ACK-VLD}}$ or $\mathcal{P}_{\text{FIFO}}$ protocol, then A and B depend on each other.

With the definition of *interface dependency*, given a multiple-block design that are connected with a set of interfaces, we can construct a dependency graph $G = (V, E)$, where G is a directed graph, V is the set of blocks in the design, each edge $(b_i, b_j) \in E$ denotes that block b_i depends on block b_j. Fig. 6.7 shows the dependency graph of the three-block design shown in Fig. 6.6. Since interface `B`, `C` and `D` are FIFOs, each block depends on the other two blocks. Therefore, there may be deadlock situations.

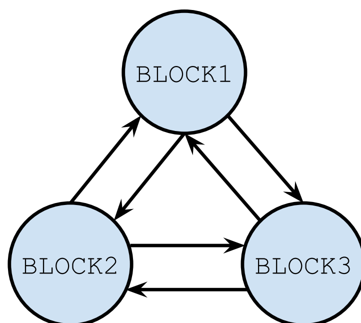


Figure 6.7: Dependency graph of Fig. 6.6.

A set $D = \{ B_1, \dots, B_m \}$ of m $(m > 1)$ blocks in a top-level design T is deadlocked if the following conditions hold:

- The inputs are available for T.
- Each block is waiting for the availability of some I/Os.

- No block can make any I/Os available while waiting.

Intuitively, at certain clock cycle, the inputs are available to the design. However, none of the blocks can make progress, and every block is waiting for other blocks to make progress in order to continue its own execution.

Let G be the dependency graph of a multi-block design. If there are any *strongly connected components* [23] in graph G, then there are deadlock possibilities in the design [65]. There are linear-time algorithms [23,30,68] to find strongly connected components of a directed graph. Note that We also need to identify a set of blocks S_i that can reach a strongly connected component S. Because some blocks in the strongly connected component S depend on blocks in S_i, progresses made in S_i may resolve the deadlock situation in S. Therefore, in order to claim that there is a deadlock in S, we must make sure that blocks in S_i are also not making progress.

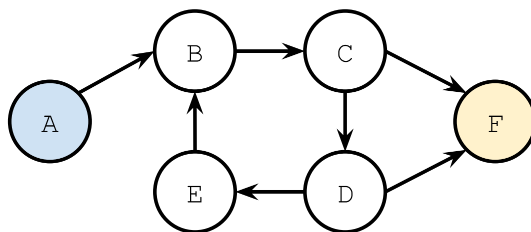


Figure 6.8: Dependency graph example

Consider an example shown in Fig. 6.8. Fig. 6.8 shows the dependency graph of a design which consists of six blocks. There is a strongly connected component { `B`, `C`, `D`, `E` }. The strongly connected component depends on block `F`. Therefore, when claiming that there is a deadlock in { `B`, `C`, `D`, `E` }, we need to consider the status of block `F`.

Now we introduce the deadlock condition generation algorithms. Let the top-level block $T = \{ b_1, \ldots, b_n \}$ contain n concurrently running sub-blocks. Algorithm 6 generates deadlock assertions from T. Condition *inputs_available* asserts that inputs of T are available. In other words, the design T deadlocks not because

Algorithm 6: GEN-DEAD-LOCK-ASSERTIONS(T)

1 $\{ b_1, \ldots, b_n \} \leftarrow T$
2 $\{ I_1, \ldots, I_k \} \leftarrow$ GET-INPUTS(T)
3 $inputs_available \leftarrow \bigwedge_{i=1}^{k}$ IS-AVAILABLE(I_i)
4 $G \leftarrow$ BUILD-DEPGRAPH($b_1, \ldots, b_n$) ▷ *build the dependency graph of blocks*
5 $Scc \leftarrow$ FIND-SCC(G) ▷ *find strongly connected components*
6 $Lock \leftarrow \emptyset$
7 **foreach** $C = \langle V, E \rangle \in Scc$ **do**
8     $V' \leftarrow$ FIND-NODES-THAT-SCC-REACHES(G, C)
9     $Lock \leftarrow Lock \cup \{ V \cup V' \}$
10 **foreach** $\{ v_1, \ldots, v_m \} \in Lock$ **do**
11     $stall_cond \leftarrow \bigwedge_{i=1}^{m}$ GENERATE-STALL-ASSERTIONS(v_i)
12   OUTPUT-ASSERTION($inputs_available \wedge stall_cond$)

Algorithm 7: GENERATE-STALL-ASSERTIONS($block$)

1 $\langle CCDFG, RTL \rangle \leftarrow block$
2 $cond \leftarrow \emptyset$
3 **foreach** $state \in CCDFG$ **do**
4   **foreach** $op \in state$ **do**
5   **if** IS-BLOCKING-IO-OPERATION(op) **then**
6             $I \leftarrow$ GET-INTERFACE(op)
7             $assertion \leftarrow (RTL.current_state = state) \wedge$ NOT-AVAILABLE(I)
8             $cond \leftarrow cond \cup assertion$
9 **return** $\bigvee_{i=1}^{|cond|} cond[i]$

of unavailability of inputs of T. Variable G is the dependency graphs of the blocks of T, and Scc contains the strongly connected components of G. For each of the

strongly connected components, we expand the node set V to include nodes that V can reach. Then *Lock* contains the expanded sets of nodes that may contain deadlocks. For each set of nodes $\{ v_1, \dots, v_m \}$ that may have deadlock, we generate stall assertions of each node v_i. *stall_cond* is a conjunction of the stall conditions of nodes $\{ v_1, \dots, v_m \}$. We then output the assertion by combining the condition *inputs_available* with stall conditions *stall_cond*.

Function GENERATE-STALL-ASSERTIONS in Algorithm 7 generates the stall condition assertions on a given block. It iterates over all the states in the CCDFG, and finds all blocking I/O operations of within a state. Function NOT-AVAILABLE asserts that the interface I is not available, and the blocking operation *op* is waiting for the availability of I. The assertion means that if the state machine of the RTL is at the state where I will be read or written but is not available, then the state machine stalls at the state, and waits for I until it becomes available. The return is a union of all stall conditions, which means the block stalls if any of the condition holds.

After generating the assertions that capture deadlock conditions, we can use simulation or formal verification tools to catch the assertions. There are commercially available mature RTL simulation tools, *e.g.*, "Incisive Enterprise Simulator" from Cadence, "Questa Simulation" from Mentor Graphics, and "VCS" from Synopsys, as well as formal verification tools, *e.g.*, "Incisive Formal Verifier" and "JasperGold Formal Property Verification App" from Cadence, "Questa Formal Verification" from Mentor Graphics, "VC Formal" from Synopsys.

6.3 EXPERIMENTAL RESULTS

We evaluated the effectiveness of the deadlock detection approach on two case studies: `DCT` and `YUV Filter`. The experiments were conducted on a workstation with Debian 7.1 running on a 2.93 GHz Intel Xeon X3470 processor with 8 GB of memory. We use "Questa Simulation" as the simulator to simulate the synthesized

RTL augmented with our generated assertions.

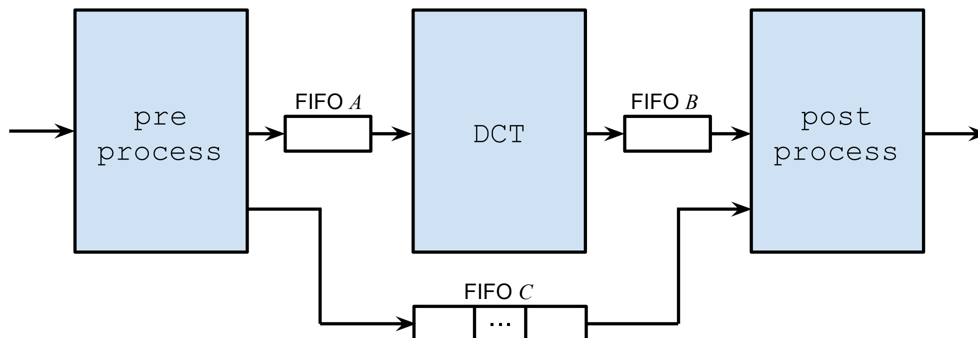


Figure 6.9: Block diagram of `DCT` example.

Fig. 6.9 shows the diagram of the `DCT` design. It consists of three blocks: block '`pre-process`' processes the input raw data; block '`DCT`' does the discrete cosine transform on the pre-processed data; and block '`post-process`' does some analysis of the data before and after the transformation. This design has 136 lines of C code, and 1966 lines of synthesized Verilog code. The depths of FIFO A and FIFO B were set to 2. We also experimented with different depths of FIFO C, the design would deadlock if the depth of C was less than 512. It took 1.93 seconds and 16.52 MB of memory to catch the deadlock assertion.

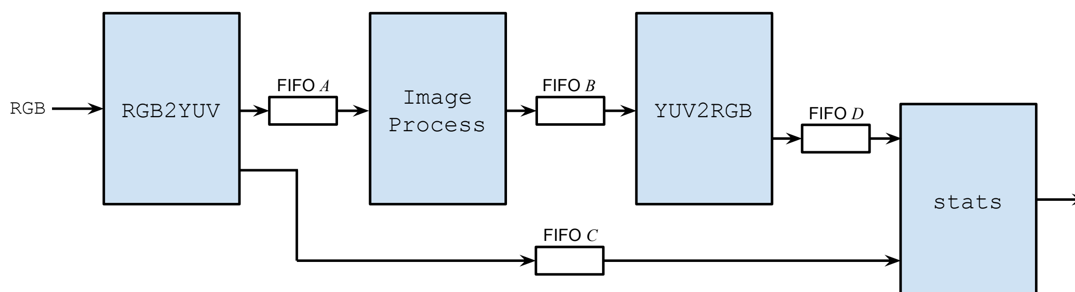


Figure 6.10: Block diagram of `YUV Filter` example.

Fig. 6.10 shows the diagram of the `YUV Filter` design, which consists of four blocks. The design first converts the input stream of RGB model to YUV model,

applies some transformation on the stream of YUV model, converts the stream back to RGB model, and does some statistics of the filter. The design has 276 lines of C code, and 5360 lines of synthesized Verilog code. The depths of the FIFOs are all set to 2. It took 46.16 seconds and 60.71 MB of memory to catch the deadlock assertion.

The two case studies all have a fork-then-join structure in the block diagrams. If the latencies are not balanced in different paths of a design, the design will deadlock. However, if the blocks have variable latencies, it is difficult to determine the existence of deadlocks statically. Deadlocks also often happen in designs with feedback loop structures. The automatically generated assertions will help the designers to detect the deadlocks either by dynamic simulation or by static formal verification tools.

6.4 RELATED WORK

In the context of distributed systems and databases, deadlocks happen whenever two or more processes are competing for limited resources. The processes are holding resources (thus preventing other processes from using them) while waiting for other resources. Deadlock detection is a well-studied problem in distributed systems and databases [9, 46]. Most approaches utilize the *wait-for* graph which captures the resource dependencies among processes. In a *wait-for* graph, nodes are used to represent processes, each edge (p, q) represents that process q is holding a resource that process p needs, therefore p is waiting for q. If there are circular waits among processes, then there are possibilities for deadlocks to happen. Graph cycle detection algorithms are often used for deadlock detection.

Deadlock detection is important for concurrent programming [8]. For example, there has been research on detecting deadlocks of concurrent Java programs [25]. This approach translates Java programs into PROMELA [39] language, and uses SPIN [40] model checker to perform formal analysis of deadlocks.

SystemC [6] is one of the most popular languages for system-level modeling. It allows the designers to model concurrency through modules and processes. Events, channels and shared variables can be used for module-level and inter-process communications. There has been research on data race detection of SystemC designs using static analysis approach [53]. One major issue with static analysis of deadlocks is that it may produce false alarms. There also has been research on formal deadlock checking of SystemC designs [18]. Symbolic simulation techniques are used to generate formulas for deadlock conditions.

6.5 SUMMARY

Behavioral synthesis generates high-performance RTL implementations by taking advantage of the concurrent nature of hardware resources. Converting a sequential high-level design into a multi-block concurrent implementation interconnected by synthesized interfaces with different handshaking protocols may introduce deadlocks. To detect the deadlocks, the designers often manually write assertions that can capture the deadlocks, and use either simulation or formal verification tools to catch the assertions. However, the synthesized RTL implementations are generally not human readable, they are intended to be consumed by downstream tools, not by designers. Therefore, manually writing sufficient assertions about deadlocks requires heavy human effort. We present an assertion-based verification approach to detecting deadlocks of the synthesized RTL implementations. Our algorithms take the design representation after the scheduling transformation and the result of the interface synthesis from the synthesis tool, automatically generate SystemVerilog assertions that can capture deadlock situations. For behavioral synthesis tool vendors, this approach can be implemented in the back-end when generating RTL implementations. Optionally, the users can instruct the behavioral synthesis tool to generate the assertions along with the RTL implementations for further verification purposes.

Chapter 7

CONCLUSION AND FUTURE WORK

7.1 CONCLUSION

Equivalence checking support is critical to the adoption of behavioral synthesis tools. In this dissertation research, we present an end-to-end scalable equivalence checking framework for the entire behavioral synthesis flow, which includes front-end compiler transformations, scheduling transformation, and back-end RTL generation. We design and develop the equivalence checking frameworks and algorithms for different phases of the behavioral synthesis flow. This dissertation makes the following specific contributions:

- Design and develop an equivalence checking framework for front-end compiler transformations via symbolically exploring paths of the source and target programs of each transformation applied.
- Develop efficient algorithms to validate scheduling transformations, including validating control and data dependencies and I/O timing of partially timed SystemC designs.
- Develop several techniques to handle design and implementation optimizations employed by behavioral synthesis tools. They are essential to the scalability of the back-end checker.
- Develop an assertion-based verification approach to detecting deadlocks introduced by concurrent RTL blocks that are interconnected by synthesized interfaces with various handshaking protocol.

According to experiments on the CHStone [37] benchmark synthesized by an open-source behavioral synthesis tool named LegUp [17], our front-end transformation equivalence checker scales to large-size designs. We are able to validate more than 75% of transformations applied by LegUp to CHStone benchmark. We evaluate our scheduling transformation validation approach on 12 synthesizable SystemC based designs in S2CBench [66] benchmark. The results show that our approach is efficient, and is able to validate the scheduling transformations on designs in S2CBench benchmark under 10 seconds. Our back-end checker handles global variables and tolerates local and irrelevant in-equivalences introduced by operation gating in the behavioral synthesis. The optimizations we developed are essential to make our back-end equivalence checker scale to real industrial-size designs. We detected a real bug in a widely used commercial behavioral synthesis tool, reported the bug to the developer. They fixed the bug in their next release. Case studies show that our assertion-based deadlock detection approach is able to catch deadlocks introduced by parallelization of RTL blocks that are connected by synthesized interfaces.

7.2 FUTURE WORK

One future direction is to extend our SEC framework to handle more aggressive transformations. The fact that we still cannot certify 25 percent of the transformations in CHStone shows that there is significant room for improvement. The future extensions include 1) equivalence checking for transformations spanning multiple procedures or functions, 2) handling transformations that modify structures of loops, perhaps through domain-specific SEC optimizations. Recall that a key reason for our inability to handle the transformations where SEC fails is the inapplicability of cut-loop, which requires equivalence for each iteration of corresponding loops of the two programs. There is a need for ways to loosen that restriction so that transformations such as partial loop unrolling can be certified.

Scheduling transformation that involves pipelining (either loop pipeline or function pipeline) is not address in this dissertation. My colleagues have done a lot of research on using equivalence checking and theorem proving techniques to verify loop pipelining [34, 60, 61] and function pipelining [35] in behavioral synthesis. Their approach is to generate a reference pipeline model by taking the pipelining parameters that are provided by the behavioral synthesis tool, and use SEC to check the equivalence between the reference model and the generated RTL. An interesting future work is to view the pipelining process as a generic transformation which manipulates the CDFG, and directly check the equivalence between the sequential CDFG and the pipelined CDFG.

For interface synthesis, currently we only focus on deadlock detection of concurrent RTL blocks interconnected by synthesized interfaces with various handshaking protocols. The correctness verification of interface synthesis has not been explored. For example, array arguments in a C function can be mapped to memory interfaces. Optimizations such as 1) partitioning an array in to multiple small memories or registers, 2) combining multiple arrays into one single memory interface, 3) changing the word-width of a memory interface are also interesting and challenging to verify.

REFERENCES

[1] *Bambu: A Free Framework for the High-Level Synthesis of Complex Applications,* `http://panda.dei.polimi.it`.

[2] IEEE Standard for Property Specification Language (PSL). *IEEE Std 1850-2005*, pages 1–143, 2005.

[3] *C-to-Silicon Compiler User Guide, 11.10*, 2011.

[4] *Catapult C Reference Manual*, 2011.

[5] *Cynthesizer Reference Guide, 4.1*, 2011.

[6] IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–1163, Jan 2012.

[7] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)*, pages 1–1315, Feb 2013.

[8] R. Agarwal, S. Bensalem, E. Farchi, K. Havelund, Y. Nir-Buchbinder, S. D. Stoller, S. Ur, and L. Wang. Detection of deadlock potentials in multithreaded programs. *IBM J. Res. Dev.*, 54(5):520–534, September 2010.

[9] Rahul Agarwal and Scott D. Stoller. Run-time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables. In *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging*, PADTAD '06, pages 51–60, New York, NY, USA, 2006. ACM.

[10] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1st edition, January 1986.

[11] D.P. Anderson and J. Ainscough. The verification of scheduling algorithms. In *IEE Colloquium on Structured Methods for Hardware Systems Design*, pages 7/1–7/5, 1994.

[12] J. Baumgartner, H. Mony, M. Case, J. Sawada, and K. Yorav. Scalable conditional equivalence checking: An automated invariant-generation based approach. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 120–127, Nov 2009.

[13] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The mathsat 4 smt solver. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 299–303, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[15] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[16] Calypto Design Systems. *Sequential Equivalence Checking: A new approach to functional verification of datapath and control logic changes*, 2007.

[17] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz 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*, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM.

[18] Chun-Nan Chou, Chang-Hong Hsu, Yueh-Tung Chao, and Chung-Yang Huang. Formal deadlock checking on high-level SystemC designs. In *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 794–799, November 2010.

[19] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: from prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, April 2011.

[20] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: from prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, April 2011.

[21] Jason Cong, Bin Liu, Rupak Majumdar, and Zhiru Zhang. Behavior-level observability analysis for operation gating in low-power behavioral synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 16(1):4:1–4:29, November 2010.

[22] Jason Cong and Zhiru Zhang. An efficient and versatile scheduling algorithm based on SDC formulation. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 433–438, New York, NY, USA, 2006. ACM.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* The MIT Press, Cambridge, Mass, 3rd edition edition, July 2009.

[24] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] Claudio Demartini, Radu Iosif, and Riccardo Sisto. A deadlock detection tool for concurrent Java programs. *Software: Practice and Experience*, 29(7):577–603, June 1999.

[26] John P. Elliott. *Understanding Behavioral Synthesis: A Practical Guide to High-Level Design*. Kluwer Academic Publishers, Boston, 1999 edition edition, May 1999.

[27] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[28] Michael Fingeroff. *High-Level Synthesis Blue Book*. Xlibris, United States, May 2010.

[29] R. Floyd. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematcs*, volume XIX, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[30] Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(34):107–114, May 2000.

[31] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, POPL '07, pages 47–54, New York, NY, USA, 2007. ACM.

[32] E. Goldberg, M. Prasad, and R. Brayton. Using SAT for Combinational Equivalence Checking. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '01, pages 114–121, Piscataway, NJ, USA, 2001. IEEE Press.

[33] Sumit Gupta, Rajesh Gupta, Nikil D. Dutt, and Alexandru Nicolau. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Springer Science & Business Media, May 2007.

[34] Kecheng Hao, Sandip Ray, and Fei Xie. Equivalence checking for behaviorally synthesized pipelines. In *Proceedings of the 49th Annual Design Automation Conference*, pages 344–349, New York, NY, USA, 2012. ACM.

[35] Kecheng Hao, Sandip Ray, and Fei Xie. Equivalence checking for function pipelining in behavioral synthesis. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6, 2014.

[36] Kecheng Hao, Fei Xie, Sandip Ray, and Jin Yang. Optimizing equivalence checking for behavioral synthesis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1500–1505, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[37] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. Proposal and quantitative analysis of the CHStone benchmark program suite for practical c-based high-level synthesis. *Information and Media Technologies*, 4(4):740–752, 2009.

[38] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, 1969.

[39] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[40] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[41] A.J. Hu. High-level vs. rtl combinational equivalence: An introduction. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 274–279, Oct 2006.

[42] Chandan 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*, 27(3):556–569, March 2008.

[43] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[44] A. Koelbl, Yuan Lu, and A. Mathur. Embedded tutorial: Formal equivalence checking between system-level models and rtl. In *Proceedings of the 2005 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '05, pages 965–971, Washington, DC, USA, 2005. IEEE Computer Society.

[45] Alfred Koelbl, Reily Jacoby, Himanshu Jain, and Carl Pixley. Solver technology for system-level to rtl equivalence checking. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 196–201, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[46] Eric Koskinen and Maurice Herlihy. Dreadlocks: Efficient Deadlock Detection. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 297–303, New York, NY, USA, 2008. ACM.

[47] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. Validating high-level synthesis. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 459–472, Berlin, Heidelberg, 2008. Springer-Verlag.

[48] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 712–717, Berlin, Heidelberg, 2012. Springer-Verlag.

[49] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.

[50] S. Malik, A.R. Wang, R.K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *, IEEE International Conference on Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers*, pages 6–9, November 1988.

[51] G. Martin and G. Smith. High-Level Synthesis: Past, Present, and Future. *IEEE Design Test of Computers*, 26(4):18 –25, August 2009.

[52] MathWorks. *HDL Coder: Generate Verilog and VHDL code for FPGA and ASIC designs, http://www.mathworks.com/products/hdl-coder/*, (accessed June 10, 2015).

[53] M. Moiseev, M. Glukhikh, A. Zakharov, and H. Richter. A static analysis approach to data race detection in SystemC designs. In *2013 IEEE 16th*

International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS), pages 54–59, April 2013.

[54] In-Ho Moon, Per Bjesse, and Carl Pixley. A compositional approach to the combination of combinational and sequential equivalence checking of circuits without known reset states. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 1170–1175, San Jose, CA, USA, 2007. EDA Consortium.

[55] Naren Narasimhan, Elena Teica, Rajesh Radhakrishnan, Sriram Govindarajan, and Ranga Vemuri. Theorem proving guided development of formal assertions in a resource-constrained scheduler for high-level synthesis. *Formal Methods in System Design*, 19(3):237–273, November 2001.

[56] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM.

[57] Rishiyur S. Nikhil. Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis*, pages 129–146. Springer Netherlands, 2008.

[58] A. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Leive, and J. Kim. The CMU Design Automation System – An Example of Automated Data Path Design. In *Design Automation, 1979. 16th Conference on*, pages 73–80, June 1979.

[59] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for*

Construction and Analysis of Systems, TACAS '98, pages 151–166, London, UK, UK, 1998. Springer-Verlag.

[60] Disha Puri, Sandip Ray, Kecheng Hao, and Fei Xie. Mechanical certification of loop pipelining transformations: A preview. In *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 549–554, 2014.

[61] Disha Puri, Sandip Ray, Kecheng Hao, and Fei Xie. Using ACL2 to verify loop pipelining in behavioral synthesis. In *Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and its Applications, Vienna, Austria, 12-13th July 2014.*, pages 111–128, 2014.

[62] David A Ramos and Dawson R. Engler. Practical, low-effort equivalence verification of real code. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 669–685, Berlin, Heidelberg, 2011. Springer-Verlag.

[63] S. Ray, W. A. Hunt, Jr., J. Matthews, and J S. Moore. A Mechanical Analysis of Program Verification Strategies. *Journal of Automated Reasoning*, 40(4):245–269, May 2008.

[64] Sandip Ray, Kecheng Hao, Yan Chen, Fei Xie, and Jin Yang. Formal verification for high-assurance behavioral synthesis. In *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis*, ATVA '09, pages 337–351, Berlin, Heidelberg, 2009. Springer-Verlag.

[65] Nicola Santoro. *Design and Analysis of Distributed Algorithms.* Wiley-Interscience, Hoboken, N.J, 1 edition edition, October 2006.

[66] B.C. Schafer and A. Mahapatra. S2CBench: Synthesizable SystemC benchmark suite for high-level synthesis. *IEEE Embedded Systems Letters*, 6(3):53–56, September 2014.

[67] Synopsys. *Synphony C Compiler High-Level Synthesis from C/C++ to RTL*,, (accessed June 10, 2015).

[68] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

[69] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 264–276, New York, NY, USA, 2009. ACM.

[70] S. Vasudevan, J. A. Abraham, V. Viswanath, and Jiajin Tu. Automatic decomposition for sequential equivalence checking of system level and rtl descriptions. In *Proceedings of the Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings.*, MEMOCODE '06, pages 71–80, Washington, DC, USA, 2006. IEEE Computer Society.

[71] K. Wakabayashi. CyberWorkBench: integrated design environment based on C-based behavior synthesis and verification. In *2005 IEEE VLSI-TSA International Symposium on VLSI Design, Automation and Test, 2005. (VLSI-TSA-DAT)*, pages 173–176, April 2005.

[72] WebM. *G2 VP9 Video Hardware RTLs, http://www.webmproject.org/hardware/vp9*, (accessed November 28, 2014).

[73] Xilinx. *Vivado Design Suite User Guide: High-Level Synthesis.*

[74] Zhenkun Yang, Kecheng Hao, Kai Cong, Li Lei, Sandip Ray, and Fei Xie. Scalable certification framework for behavioral synthesis front-end. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 149:1–149:6, New York, NY, USA, 2014. ACM.

[75] Anna Zaks and Amir Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *Proceedings of the 15th International Symposium on Formal Methods*, FM '08, pages 35–51, Berlin, Heidelberg, 2008. Springer-Verlag.

[76] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. pages 427–440, 2012.

[77] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: a methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9:2003, 2003.