ABSTRACT

An abstract of the thesis of Yan Chen for the Master of Science in Computer Science presented July 8, 2008.

Title: Equivalence Checking for High-Level Synthesis Flow

High-level synthesis provides a promising solution to design complicated circuits, but the lack of designers' confidence in correctness of synthesis tools prevents the wide acceptance in engineering practice. I develop an equivalence checking algorithm within a framework for certifying high-level synthesis flow. I utilize a new formal structure *clocked control data flow graph* (CCDFG) to facilitate the equivalence checking process, and implement the prototype tool for verifying the equivalence between CCDFG and synthesized circuit in both bit-level and word-level. Experimental results demonstrate the effectiveness of the tools in quickly verifying, or finding bugs in the high-level synthesis flow.

EQUIVALENCE CHECKING FOR HIGH-LEVEL SYNTHESIS FLOW

by

YAN CHEN

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

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
2008

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

With the rapid miniaturization of VLSI technology, it is becoming increasingly challenging to develop reliable, high-quality systems that make effective use of all the available transistors. The complexity of modern circuits makes it infeasible to develop reliable hand-crafted designs at gate level or even register-transfer level (RTL). High-Level Synthesis [2, 6, 10, 14] (HLS) provides a promising solution to this problem, namely automated synthesis of the design from a behavioral specification. The behavioral specification is written in a high-level language such as SystemC or C; a synthesis tool applies a sequence of transformations to compile this description into a hardware netlist or RTL.

In spite of its promise, high-level synthesis has not yet found wide acceptance in engineering practice [14]. A major barrier is the lack of designers' confidence in correctness of synthesis tools. The large semantic gap between the synthesized design and its behavioral description puts the onus on synthesis to ensure that its output conforms to the specification. On the other hand, the employed transformations include complex optimizations to satisfy diverse performance and power metrics, making synthesis tools error-prone.

My motivation is that a scalable, mechanized framework for certifying high-level synthesis flow will help designers be confident about the synthesized circuits, and making the synthesis tools more reliable. In particular, an efficient equivalence checking algorithm for high-level synthesis can guarantee the correctness of the circuit even after some minor manual optimizations.

1.1 RESEARCH OVERVIEW

My research is mainly focused on the equivalence checking of high-level synthesis flow. This work builds directly on the work of Sandip Ray, Fei Xie and me on a framework for certifying high-level synthesis flow.

Certification of the high-level synthesis flow is decomposed into verified and verifying components, which are discharged by theorem proving and model checking respectively. The bridge between these components is provided by a new formal structure, *clocked control data flow graph* (CCDFG), that serves as the golden circuit model used in this framework.

After theorem proving verified the CCDFG, my work analyzes the CCDFG by symbolic simulation and tries to prove that the verified CCDFG and the synthesized circuit are equivalent. If they are not equivalent, the verification tool reports the mismatched points, which is useful for the user to find the bugs.

In this thesis, my equivalence checker uses symbolic simulation in both bit-level and word-level. By bit-level, I use the Binary Decision Diagram (BDD) [5] to bit-blasting all the data into bits. This approach may have space explosion problems, but theoretically it can verify all the computation being considered. To scale the model checking process, I also consider using word-level abstraction, mainly uninterpreted functions, bit vectors and linear arithmetic, to handle much more complex designs. Although this approach has some limitations due to the undecidability natural of Satisfiability Modulo Theory (SMT) [3], it is able to verify many practical designs, which demonstrate the practical usefulness of my approach.

1.2 THESIS OUTLINE

I will give a brief overview of background in Chapter 2 that introduces high-level synthesis flow and the overall framework of the certification. I will also give the formal semantics of CCDFG, which will be used as the golden circuit model in the equivalence

checking, and discuss some related work on sequential equivalence checking. In Chapter 3, I will describe the challenges for my verification methods and detailed solutions to them. To test the effectiveness, I apply my tools to two real world examples, and demonstrate the capability of my apporach. Chapter 4 goes into the conclusion, and some future work.

Chapter 2

BACKGROUND

2.1 HIGH-LEVEL SYNTHESIS

High-level synthesis [14] is the process of converting a behavioral description, usually written in a high-level language such as C or SystemC, into a digital circuit that consists of a data path, a controller and memory elements. The first task in high-level synthesis is to capture the behavioral description in an intermediate representation that captures both control flow and data flow. Thereafter, the high-level synthesis problem has usually been solved by dividing the problem into several sub-tasks. Typically the subtasks are:

- **Allocation:** This task consists of determining the number of resources that have been allocated or alloted to synthesize the hardware circuit. Typically, designers can specify an allocation in terms of the number of resources of each resource type. Resources consist not only of functional units (like adders and multipliers), but may also include registers and interconnection components (such as mux and bus).

- **Scheduling:** The scheduling problem is to determine the time step or clock cycle in which each operation in the design executes. The ordering between the "scheduled" operations is constrained by the data and control dependencies between operations. Scheduling is often done under constraints on the number of resources as specified in resource allocation.

- **Module Selection:** Module selection is the task of determining the resource type from the resource library that an operation executes on. The need for this task

arises because of the presence of several resources of different types (and different area and timing) that an operation may execute on. For example, an addition may execute on an adder, an ALU, or a multiply-accumulate unit. There are area, performance, and power trade-offs in choosing between different resources such that a metric is minimized.

- **Binding:** Binding determines the mapping between the operations, variables and data (and control) transfers in the design and the specific resources in the resource allocation. Hence, operations are mapped to specific functional units, variables to registers, and data/control transfers to interconnection components.

- **Control Generation and Optimization:** Control synthesis generates a control unit that implements the schedule, usually represented as a finite state machine. This control unit generates control signals that control the flow of data through the data path (i.e. through the mux). Control optimization tries to minimize the size of the control unit and hence, improve metrics such as area and power.

Each of these tasks are interlinked and often dependent on each other. As they may contain complicated heuristics, many things can go wrong during the synthesis process. The motivation of my research is trying to check the equivalence of the intermediate representation and the synthesized circuit.

2.2 FRAMEWORK FOR CERTIFICATION OF HIGH-LEVEL SYNTHESIS FLOW

Certification of a synthesis flow amounts to the guarantee that its output preserves the semantics of its input description; thus, the question of correctness of synthesized designs is reduced to the question of analysis of the behavioral specification.

The analysis of a practical synthesis flow is non-trivial for several reasons. The transformations performed by a synthesis tool include (1) “generic compilation steps” such

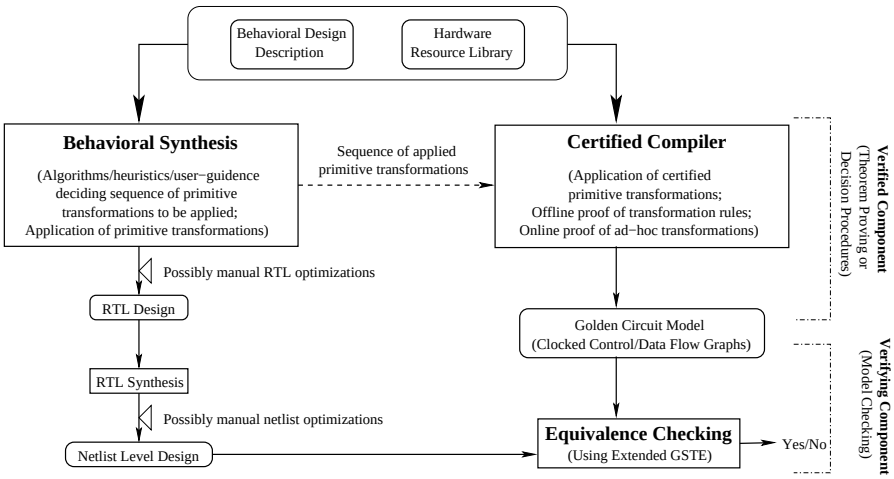

Figure 2.1: Framework for Analysis of High-Level Synthesis Flow

as *loop unrolling*, *code motion*, and *common subexpression elimination*, (2) "scheduling transformations" that order operations to meet the available resource constraints, (3) "optimizing transformations" to achieve the overall target metric of area, power, and efficiency, and (4) "ad hoc and manual transformations" or "tweaks" are often inserted to fine-tune the output of the automated flow to specific design metrics. A transformation may depend implicitly on complex invariants of other transformations in the overall flow.

The overall analysis framework is shown in Fig. 2.1. In summary, we decompose analysis of transformations into two components, *verified* and *verifying*.[1] A *verified* transformation is formally proven once and for all to preserve the semantics of its input. The proof is done offline and discharged by a theorem prover. This is suitable for transformations associated with generic and reusable compilation steps at the higher levels of the synthesis flow, where the cost of theorem proving is mitigated by reusability of the transformations. A *verifying* transformation is not itself verified, but each instance

[1]The terms "*verified" and "*verifying" as used here have been borrowed from analogous notions in the compiler certification literature [20, 1].

of its application is accompanied by a justification of correspondence. Since the obligations are discharged for each instance, the verification must be automatic; the verifying component is implemented by model checking. I will discuss the verifying component in more detail in Chapter 3.

The framework requires a smooth interface between the *verified* and *verifying* components. This interface is provided by a formal graph-based design representation, namely *clocked control data flow graph* (CCDFG). The CCDFG for a design can be derived from its behavioral description and can be viewed as the formal representation of the golden circuit model. A CCDFG can be viewed as a formal rendition of control data flow graph (CDFG) — used as an intermediate representation in many synthesis tools — augmented with the notion of a schedule. Compiler transformations are viewed as transformations of CCDFG, and the *verified* component involves proof that the CCDFG generated by a certified transformation is a refinement of its input CCDFG. To facilitate such verification, Sandip formalized the execution semantics of CCDFG in the ACL2 theorem prover and developed a formal notion of refinement based on execution correspondence. Theorem proving facilitates the proof of *generic* properties that can certify large classes of similar transformation in one swoop. The explicit representation of control and data flow enables definition of invariants without augmenting the model with auxiliary flow information; scheduling enables the use of CCDFGs for both pre-scheduling and in-scheduling transformations. The transformed CCDFG is used by the *verifying* component to derive correspondence with the synthesized circuit. My equivalence checking paradigm leverages the high capacity of GSTE-style model checking [22] and the cycle-accurate nature of CCDFGs. The equivalence checking is conducted as a dual-rail symbolic simulation, with the upper rail being the simulation of the CCDFG and the lower rail being simulation of the circuit implementation. The two rails are synchronized by clock cycle.

2.3 SEMANTICS OF CCDFG

In this section, I give the semantics of CCDFG. Since a CCDFG is derived from a CDFG, we first review the notion of CDFG; we then formalize CCDFG and define its execution semantics.

Remark. (Input Language Assumptions). We leave the underlying input language unspecified, with the following "well-formedness" assumptions. The language is assumed to provide a partition of design variables into *state variables* and *input variables*. The legal expressions in the language are generated by a well-defined grammar over the state and input variables and language constants; given a mapping of the variables to constants, any legal expression is computable. Each instruction in the language can be decomposed into a sequence of *primitive operations*; the set of operations includes standard arithmetic and logical operations, comparisons, assignments, etc. together with standard if-then-else and loop constructs. Designs specifications in the language are assumed to be amenable to usual control and data flow analysis. The control flow is broken up into a number of *basic blocks*, each with a single entry and exit. Data dependency is given by a "read after write" paradigm: an operation op_j is data dependent on an operation op_i if op_j occurs after op_i in some control flow path and computes an expression over some state variable v that is assigned most recently by op_i in the path. The language is assumed to disallow circular data dependencies.

Definition 2.1 (Control Flow and Data Flow Graphs)**.** Let $ops \triangleq \{op_1, \ldots, op_n\}$ be a set of operations over some set V of (state and input) variables, and bb be a set of basic blocks each consisting of a sequence of operations over *ops*. A *data flow graph* G_D over *ops* is a directed acyclic graph such that each vertex of G_D is a member of *ops*. A *control flow graph* G_C is a labeled graph with *bb* being the set of vertices and each edge labeled with a Boolean assertion over V.

An edge in G_D from op_i to op_j represents a data dependency from op_i to op_j, and an edge in G_C from bb_i to bb_j indicates that bb_i is a direct predecessor of bb_j in the control

flow structure of the program. An assertion along an edge is a predicate that must hold whenever program control makes the corresponding transition.

Definition 2.2 (CDFG)**.** Let $ops \triangleq \{op_1, \ldots, op_m\}$ be a set of operations over a set of variables V, $bb \triangleq \{bb_1, \ldots, bb_n\}$ be a set of basic blocks over ops, G_D and G_C are data and control flow graphs over ops and bb respectively. A ***CDFG*** is the tuple $G_{CD} \triangleq \langle G_D, G_C, H \rangle$, where H is a mapping $H : ops \rightarrow bb$ such that $H(op_i) = bb_j$ iff op_i occurs in bb_j.

The order of execution of operations in a CDFG is irrelevant as long as the control and data dependencies are respected. The definition of *microsteps* below makes this notion explicit.

Definition 2.3 (Microstep Ordering and Partition)**.** Let $G_{CD} \triangleq \langle G_C, G_D, H \rangle$, where the set of vertices of G_C is $bb \triangleq \{bb_1, \ldots, bb_l\}$, and the set of vertices in G_D is $ops \triangleq \{op_1, \ldots, op_n\}$. For each $bb_k \in bb$, a ***microstep ordering*** is a relation $\prec_k$ over $ops(bb_k) \triangleq \{op_i : H(op_i) = bb_k\}$ such the $op_a \prec_k op_b$ if and only if there is a path from op_a to op_b in the subgraph $G_{D,k}$ of G_D induced by $ops(bb_k)$. A ***microstep partition of*** bb_k under $\prec_k$ is a partition M_k of $ops(bb_k)$ satisfying the following two conditions. (1) For each $p \in M_k$, if $op_a, op_b \in p$ then $op_a \not\prec op_b$ and $op_b \not\prec_k op_a$. (2) If $p, q \in M_k$ with $p \neq q$, $op_a \in p$, $op_b \in q$, and $op_a \prec_k op_b$, then for each $op_{a'} \in p$ and $op_{b'} \in q$ $op_{b'} \not\prec_k op_{a'}$. A ***microstep partition of*** G_{CD} is a set M containing each microstep partition M_k.

Since G_D is acyclic, $\prec_k$ is an irreflexive partial order on $ops(bb_k)$ and the notion of microstep partition above is well-defined. Given a microstep partition $M \triangleq \{m_0, m_1, \ldots\}$ of G_{CD} each m_i is called a ***microstep*** of G_{CD}. It is convenient to view $\prec_k$ as a partial order over the microsteps of bb_k, and further extend it without loss of generality to a total order. Informally, if op_a and op_b are in the same partition, their order of execution does not matter; if p and q are two microsteps where $p \prec_k q$, the operations in p must be executed before q to respect the data dependencies.

Remark. We formalize the execution of a computing model as a sequence of the underlying design states under a legal input sequence; given a state and legal input, the semantics specifies the next state. For a CDFG (and CCDFG), states and inputs are the valuation of the state and input variables; for circuit models (cf. Section 3.1), states correspond to the valuation of latches and inputs to the valuation of input signals. When the underlying model is clear, we use the terms "state" and "input" without qualification; when discussing correspondence between two different models we make the model explicit, for instance referring to "CCDFG states" and "circuit states".

In the following definition, we leave the result of executing individual operations unspecified, but assume that it can be derived from the input language. The result of executing a microstep m_j from state s under input i is a computable function f_j that computes the valuation of the state variables updated by the constituent operations; since there is no data dependency among these operations, the order of evaluation does not matter.

Definition 2.4 (Execution Semantics of CDFG)**.** Given a CDFG, G_{CD}, a microstep partition M of G_{CD}, and a sequence of inputs $i_0, i_1, \ldots$, an execution of G_{CD} is a state sequence, $\mathcal{E} \triangleq s_0, s_1, \ldots$ satisfying the following conditions. (1) There exists a sequence of microsteps $\mathcal{P} \triangleq m_0, m_1, \ldots$ of G_{CD} such that s_{j+1} is the result of executing m_j from state s_j under input i_j. (2) if $m_j, m_{j+1} \in bb_k$, then (i) $m_{j+1} \not\prec_k m_j$, and (ii) there is no $p \in bb_k$ such that $m_j \prec_k p$ and $p \prec_k m_{j+1}$. (3) If $m_j \in bb_k$ and $m_{j+1} \in bb_l$, $k \neq l$, then (i) for each $p \in bb_k$ and $q \in bb_l$ $m_j \not\prec_k p$ and $q \not\prec_l m_{j+1}$, and (ii) there is an edge e in G_c from bb_k to bb_l, and (iii) the assertion on e evaluates to true under state s_j and input i_j. We call $\mathcal{P}$ the *inducing sequence* of $\mathcal{E}$.

We now formulate CCDFG, by augmenting a CDFG with a schedule. Consider a microstep partition M of G_{CD}. A *schedule* T of M is a partition or *grouping* of M; for $m_1, m_2 \in M$, if m_1 and m_2 are in the same group in T, we say that m_1 and m_2 belong to the same scheduling step.

```
int gcd (int a, int b)
{
  do {
    if (a < b) swap (a, b);
    a = a % b;
  } while (a != 0);
  return b;
}
```

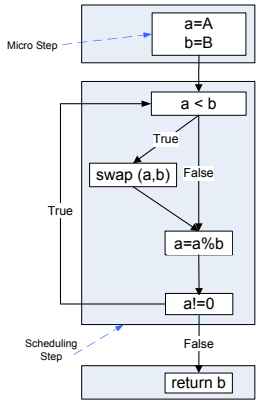

Figure 2.2: Source code for GCD and the corresponding CCDFG. In the CCDFG, each while box denotes a micro step and each shaded region denotes a scheduling step. The primitive operations here are assignment, comparison, modular division, and swap. To simplify presentation, only control dependency edges are shown and data dependency edges are omitted.

Definition 2.5 (CCDFG)**.** A *CCDFG* is a tuple $G \triangleq \langle G_{CD}, M, T \rangle$, where G_{CD} is a CDFG, M is a micro-step partition of G_{CD}, and T is a schedule of M.

Fig. 2.2 shows the relation between a high-level GCD program and a corresponding CCDFG. Note that the CCDFG corresponds closely to the high-level description.

We need the following two definitions for CCDFG execution semantics. The first formalizes the criterion for a sequence of microsteps to respect a schedule. The second formalizes the notion that the inputs at the same scheduling step are fixed.

Definition 2.6 (Microstep Sequence Consistency)**.** Let M be a microstep partition of a CDFG, T be a schedule of M, $\mathcal{P} \triangleq m_0, m_1, \ldots$ be a sequence of microsteps of M, and N be a mapping that assigns a natural number to each microstep in $\mathcal{P}$. We say that $\mathcal{P}$ is *consistent* with T under N if the following conditions hold. (1) for $m_i, m_j \in \mathcal{P}$ if $i < j$ then $N(m_i) \leq N(m_j)$; and (2) if $N(m_j) = N(m_{j+1})$ then m_j and m_{j+1} belong to the same group under T.

We say that N is a *witness* to consistency of $\mathcal{P}$. A microstep sequence $\mathcal{P}$ is *consistent*

with T if there is a mapping N such that $\mathcal{P}$ is consistent with T under N.

Definition 2.7 (Input Sequence Conformance)**.** Let M be a microstep partition of a CDFG, T be a schedule of M, and $\mathcal{P} \triangleq m_0, m_1, \ldots$ be a sequence of microsteps from M that is consistent with T under a witness N. Then an input sequence $i_0, i_1, \ldots$ is *conformant* with $\mathcal{P}$ under N and T if, for each j such that $i_j \neq i_{j+1}$, $N(m_{j+1}) = N(m_j) + 1$.

We now formalize the semantics of CCDFG execution.

Definition 2.8 (Execution Semantics of CCDFG)**.** Let $G \triangleq \langle G_{CD}, M, T \rangle$ be a CCDFG, and $\mathcal{P}$ be a sequence of microcode consistent with T under a witness N. Then $\mathcal{E} \triangleq s_0, s_1, \ldots$ is an *execution* of G if the following hold. (1) $\mathcal{E}$ is an execution of G_{CD} corresponding to some input sequence $\mathcal{I} \triangleq i_0, i_1, \ldots$. (2) $\mathcal{P}$ is an inducing sequence of $\mathcal{E}$. (3) $\mathcal{I}$ is conformant with $\mathcal{P}$ under N and T.

Thus each execution of a CCDFG is an execution of the underlying CDFG but not vice versa; the conformance requirement restricts the sequence of legal inputs and hence executions.

Finally, we consider *outputs* and *observation*. An *output* of a CCDFG G is some computable function f of (a subset of) state variables of G; informally, f corresponds to some output signal in the circuit synthesized from G. For each state s of G, the *observation* corresponding to an output f at state s is the valuation of f under s. Given a set F of output functions, any sequence $\mathcal{E}$ of states of G induces a sequence of observations $\mathcal{O}$; we refer to $\mathcal{O}$ as the *observable behavior* of $\mathcal{E}$ under F.

2.4 CERTIFIED COMPILATION

The *verified* component of the framework, namely certification of transformations, uses theorem proving to certify generic transformations applied by high-level synthesis flow. To give readers a complete picture of the framework, I give a transformation example to

show the process of this component. Since it is not the main focus of my thesis, I omit the technique details here.

Informally, the goal in certifying that a high level synthesis transformation is to show that applying it on CCDFG G results in a refinement of G. However, we must additionally account for the possibility that a transformation may be applicable to G only if G has a specific structural characteristic; furthermore the result of application might produce a CCDFG with a characteristic that facilitates the subsequent application of another transformation. To make explicit the notion of applicability of a transformation on a CCDFG, it is convenient to view a transformation as a "guarded command" [15, 12] $\tau \triangleq \langle pre, \mathcal{T}, post \rangle$. Informally, τ is applicable to a CCDFG which satisfies *pre* and produces a CCDFG which satisfies *post*.

Consider the sequence of transformations shown in Fig. 2.3 for our GCD example. The transformed code conducts two modular operations in one cycle, thus speed up the computation of GCD. Once we can discharge a transformation into some primitive transformations as shown in Fig. 2.3, each primitive transformation can be proof by theorem proving. The transformations applied in the theorem proving part may affect the performance of model checking.

2.5 RELATED WORK

There has been much research on sequential equivalence checking (SEC) between RTL and gate-level hardware designs [4, 17]. Research has also be done on combinational equivalence checking between high-level designs in software-like languages (e.g., SystemC) and RTL-level designs [16].

There has also been effort for SEC between software specifications and hardware implementations [13]: GSTE assertion graphs were extended so that an assertion graph edge have pre and post condition labels, and also associated assignments that update state variables. These extended assertion graphs motivated our formulation of CCDFGs,

```
/* Original */
do {
  if (a < b)
    swap (a, b);
  a = a % b;
} while (a != 0);
return b;
```

```
/* Transformation 1 */
do {
  if (a < b)
    swap (a, b);
  a = a % b;
  if (! (a != 0))
    return b;
  if (a < b)
    swap (a, b);
  a = a % b;
} while (a != 0);
return b;
```

```
/* Transformation 2 */
do {
  swap (a, b);
  a = a % b;
  if (! (a != 0))
    return b;
  swap (a, b);
  a = a % b;
} while (a != 0);
```

```
/* Transformation 3 */
do {
  a = a % b;
  if (! (a != 0))
    return b;
  b = b % a;
  if (! (b != 0))
    return a;
} while (1);
```

Figure 2.3: An example of certifiable transformation sequence. The sequence includes (1) unrolling the loop once, (2) interpreting the "%" operation to show that (a < b) holds after the assignment a = a % b, and (3) loop transformation through interpretation of swap operation. Due to space limitation, we use C code instead of the CCDFGs to represent the transformation.

which preserve both control/data flows and the scheduling information. My scheme is analogous to but extends traditional GSTE-style model checking [23] in the following sense. Like GSTE, the simulation is guided by a graph, namely the CCDFG, and the simulation complexity largely depends on that of the graph since the fixed-point computation is conducted on the CCDFG and only the current circuit state is kept; on the other hand, CCDFGs provide richer information than *assertion graphs* employed by GSTE, with explicit specification of valuation of state variables which can be symbolically simulated to generate state sequences. In contrast, assertion graph edges are only labeled with preconditions and postconditions. Also GSTE can only check properties in bit-level, while my approach can run in both bit-level and word-level. There is also much research on symbolic simulation of software [19], hardware [18] and embedded systems [11] in word-level.

There has also been work on equivalence checking with other graph representations, e.g., Signal Flow Graph (SFG) [9].

Chapter 3

EQUIVALENCE CHECKING FOR HIGH-LEVEL SYNTHESIS

The basic goal for my verification method is to check the equivalence between a verified CCDFG and a synthesized circuit. I first formulate the notion of equivalence between a CCDFG and a circuit. Then I discuss my approach for checking the equivalence in bit-level. Finally, I consider word-level abstractions to scale the capacity of my equivalence checker.

3.1 CIRCUIT MODEL

I represent a circuit as a Mealy machine specifying the updates to the state elements (latches) in each clock cycle. My formalization of circuits is typical in traditional hardware verification, but we make combinational nodes explicit to facilitate correspondence with CCDFGs.

Definition 3.1 (Circuit)**.** A circuit is a tuple $M_C = \langle I, N, F \rangle$ where I is a vector of input signals; N is a pair $\langle N_c, N_d \rangle$ where N_c is a set of *combinational nodes* and N_d is a set of *latches*; and F is a pair $\langle F_c, F_d \rangle$ where F_c maps each combinational node $c \in N_c$ to a Boolean expression over $N \cup I$. and for each latch $d \in N_d$, F_d maps each latch d to a node $n \in N$ where F_d is a delay function which takes the current value of n to be the next-state value of d.

A *circuit state* is an assignment to the latches in N_d; we assume a pre-assigned *initial state*, corresponding to the values of the latches at reset. Given a circuit state and a valuation of the input signals I, we compute the *circuit transition* at each clock

cycle as follows. The output of each combinational node $c \in N_c$ is the valuation of the function $F_c(c)$ on the current circuit state and the input valuation; the next state of each latch $d \in N_d$ is the valuation of $F_d(d)$. Combinational nodes are updated at the beginning of a clock cycle and the latches are updated at the end; the state updates are thus delayed to reflect propagation of signals through circuit wires.

We now formalize circuit executions. Given a sequence of valuations to the input signals $i_0, i_1, \ldots$, a *circuit trace* of M is the sequence of states $s_0, s_1, \ldots$, where (1) s_0 is the initial state and (2) for each $j > 0$, the state s_j is obtained by updating the elements in N_d given the state valuation s_{j-1} and input valuation i_{j-1}.

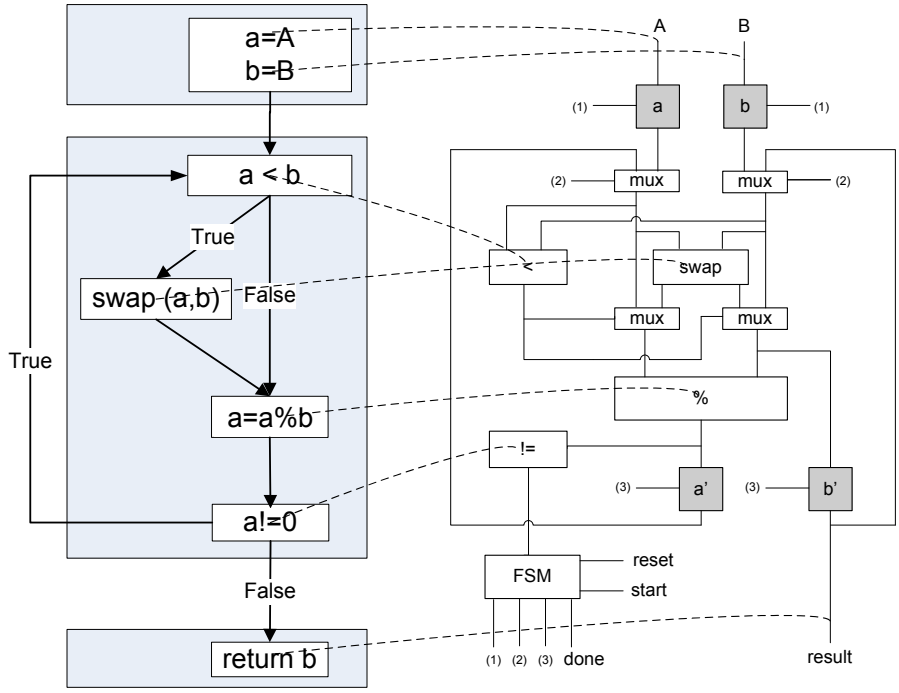

Figure 3.1: Synthesized circuit for GCD and its operation mapping relation with CCDFG in Fig. 2.2. $A, B, reset, start$ are the input nodes, and the shaded nodes a, a', b, b' are latches. The dotted lines represent the mapping from operations of CCDFG to combinational nodes of circuit. We consider a wire to be a combinational node.

Fig. 3.1 shows a synthesized circuit derived from the CCDFG in Fig. 2.2. Note that

FSM is the control component of the circuit, which contains both combinational nodes and latches. Given any circuit state, FSM will decide all control signals for the circuit, and finally when computation finishes, the result will be available in $result$, and $done$ signal is set to true.

3.2 CORRESPONDENCE BETWEEN CCDFGS AND CIRCUITS

Given a CCDFG G and a synthesized circuit M_C, how do we define execution correspondence? Note that we can define a natural mapping between the inputs of G and the input signals of M_C. It is thus tempting to define the correspondence between G and M_C as follows: (1) establish a mapping between the state variables of G and the latches in M_C, and (2) stipulate an execution of G to be equivalent to an execution of M_C if they have the same sequence of observable behaviors.

However, equivalence based on fixed mappings of variables does not work in general. Although there are fixed mappings between input variables in the CCDFG and input signals of the circuit, the mappings between internal variables and latches may be different in each clock cycle. We address this by introducing mappings between the CCDFG *operations* and the combinational nodes in the circuit: each operation op is mapped to the set of combinational nodes that together implement op; note that this mapping is independent of clock cycles.

We formalize these mappings by the definitions of $IMap$ and $NMap$ below. Suppose G is a CCDFG with a set of input variables V_I and a set of operations *ops*, and $M_C \triangleq \langle I, N, F \rangle$ is a circuit. Then $IMap : V_i \rightarrow I$ is a one-to-many mapping from the input variables of G to the input signals of M; for each input variable of v of G, $IMap(v)$ returns the corresponding set of input signals of M_C. Finally, $NMap : ops \rightarrow N_c$ is a mapping from the operations of G to the combinational nodes of M_C, which determines how each operation is implemented in M_C. For the GCD example, since variables a, b are the input of CCDFG, we define $IMap(a) = A$ and

$IMap(b) = B$ (cf. Fig. 3.1). Each CCDFG operation corresponds to a combinational node.

We now define the equivalence between a CCDFG state of G and a circuit state of M_C with respect to a given scheduling step of G and under the equivalent inputs.

Definition 3.2. A CCDFG state x of G is equivalent to a circuit state s of M_C with respect to an input i and a microstep partition M, if for each operation op in t, the inputs to op according to x and i are equivalent to the inputs to $NMap(op)$ according to s and $IMap(i)$, i.e., the values of an input to op and the corresponding input to $NMap(op)$ are equivalent, and the outputs of op are equivalent to the outputs of $NMap(op)$. Given a CCDFG G and a circuit M_C, G is equivalent to M if and only if for any execution $[x_0, x_1, x_2, \ldots]$ of G that is generated by an input sequence $[i_0, i_1, i_2, \ldots]$ and by the execution $[t_0, t_1, \ldots]$ of G, and state sequence $[s_0, s_1, s_2, \ldots]$ of M generated by the input sequence $[IMap(i_0), IMap(i_1), IMap(i_2), \ldots]$, x_k and s_k are equivalent with respect to t_k under i_k, $k \geq 0$.

Note that the initial states x_0 and s_0 of G and M_C are irrelevant in that the operations in the first scheduling step of G (or, respectively, the corresponding circuit nodes of M under $IMap$, respectively) depend on i_0 (or $IMap(i_0)$), but not x_0 (or s_0). Therefore, x_0 and s_0 can be arbitrary while the requirement that v_0 and s_0 are equivalent with respect to t_0 under i_0 are still satisfied.

Finally note that, not all combinational nodes in the circuit have their corresponding operations in CCDFG. For example, the mux nodes and FSM in the circuit are not represented in the CCDFG. These unobservable parts constitute the control component generated by synthesis to preserve the control and data dependencies of the CCDFG, and their correctness is implied by the equivalence of observable nodes.

3.3 DUAL-RAIL SIMULATION FOR SEQUENTIAL EQUIVALENCE CHECKING

To check the above equivalence between a CCDFG, G, and a circuit, M, we propose a dual-rail symbolic simulation scheme shown in Fig. 3.2.

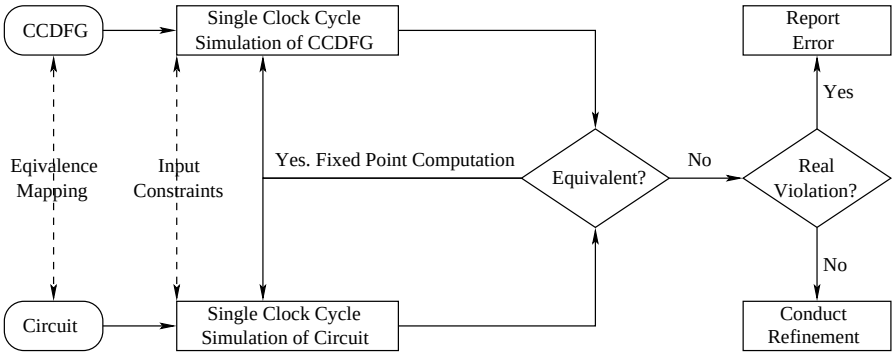

Figure 3.2: Dual-Rail Simulation Scheme for Equivalence Checking

The upper rail simulates G while the lower rail simulates M_C. The two rails are synchronized by clock cycle, and follow an abstraction/refinement paradigm. The equivalence checking scheme for clock cycle k can be roughly summarized as follows:

1. The current CCDFG state x_k and circuit state s_k are checked to see whether for the input i_k, the inputs to each operation op in the scheduling step t_k are equivalent to the inputs to its corresponding circuit nodes in $NMap(op)$. If the inputs are equivalent, go to Step 2; otherwise, go to Step 3.

2. G is simulated by executing t_k on x_k under i_k to obtain state x_{k+1}, recording the outputs of each $op \in t_k$. Correspondingly, M is simulated for one clock cycle from s_k under the input $IMap(i_k)$ to obtain circuit state s_{k+1}. The outputs of each op are checked for equivalence under $NMap$ against the outputs of the nodes in $NMap(op)$. If the inputs are equivalent, go to Step 4; otherwise, go to Step 3.

3. We check if the failure on equivalence check is a false negative caused by abstraction. If there is no false negative, the problem is reported; otherwise, the abstraction is refined. The report is associated with an error trace that contains the CCDFG state sequence, the execution of G, and the circuit state sequence of M_C up to the current clock cycle.

4. The scheduling step t_{k+1} is determined based on the control flow. If t_k has multiple outgoing control edges, the last microstep of t_k executed in the simulation above is identified. The outgoing control edge from this micro-step whose condition evaluates to be true leads to t_{k+1}.

The simulation proceeds cycle-by-cycle until either (i) the equivalence check fails, or (ii) a fixed point is reached and there is no observable inconsistency between CCDFG and circuit. If there is a mismatch during the equivalence checking for a certain operation. There may be two possible causes either (i) the synthesized operation is not correctness, possibly by choosing the wrong bit width for that operation, or (ii) the FSM is not correctly implemented, so the data is going through the wrong data path. In all, my mapping relation can capture synthesize errors in both data flow and control flow.

3.4 OPTIMIZATION THROUGH WORD-LEVEL ABSTRACTION

Among the research on equivalence checking of sequential circuits, Binary Decision Diagrams (BDD) [5] are a very successful representation for model checking on many practical systems. However, BDD-based method cannot scale well to verify very large designs. That is mainly because, to conduct equivalence checking using BDDs, we need to do two things.

1. All the variables in CCDFG should be represented in bit-level. As a result, the performance of model checking largely depends on the bit width of the design being considered. The more bit width we are trying to verify, the more variables we need to use for BDDs.

2. All high-level operations, such as addition and multiplication, need to be implemented in bit-level. For some functions like multiplication, the size of BDDs can grow exponentially in the number of variables being considered.

This motivates the development of tools which can operate at a higher level of abstraction. In this thesis, I use word-level abstraction to represent the states in CCDFG, and borrow a Satisfiability Modulo Theories (SMT) solver CVC3 [3] as the decision procedure.

CVC3 is a SMT solver that supports the theories of linear arithmetic, bit vectors and uninterpreted functions. Unfortunately, some complicated computations, such as nonlinear arithmetic, is undecidable in general. In order to solve this problem, I have to make a simplifying assumptions: the control flow must be analyzable — the designer can give a iteration bound for all loops. Otherwise, my equivalence checker can only check whether a CCDFG and a circuit is equivalent up to a fixed bound. To the best of my knowledge, many industry design have bounded loops, so this simplification does not reduce too much practical usefulness of my approach.

Furthermore, I assume CCDFG and the synthesized circuit have very similar control flow. As we are simulating CCDFG in word-level, we also need to abstract the circuit into word-level. As a first step, I build a CCDFG for the circuit based on its VHDL implementation, and then check whether the two CCDFGs are equivalent in word-level. If the control structure is too different, we cannot verify the equivalence due to the imprecision introduced by word-level abstraction.

3.4.1 Uninterpreted Functions

Under the above simplifications, my equivalence checker use uninterpreted functions to handle undecidable part of the operations. An uninterpreted function is a function that knows nothing other than its name and it is a function in mathematical sense (it does not have side-effects). For example, if we know that $a = x$ and $b = y$, then we know

$f(a, b) = f(x, y)$, regardless of what the function f is and what data type of a, b, x and y are.

Uninterpreted functions naturally abstract away the details of hardware data path operations. This abstraction makes a lot of sense in the high-level synthesis flow, since we care more on the higher-level problem of what operations are being applied to what operands, instead of whether a multiplier actually multiplies. This abstraction is particularly useful when some data path does not effect the control flow of the program. In this case, we do not need to evaluate the results without losing any information.

The abstraction is also conservative — the abstraction will not cause two inequivalent designs to be declared equivalent — but it is sometimes too conservative. For example, if in CCDFG we have $a \times 2$, and in the synthesized circuit, the operation becomes left shifting a by one bit. Although these two operations are equivalent for bit vector a. But uninterpreted functions can not handle this case. The good thing is that since we have the mapping relation of operations between CCDFG and circuit, we can pre-certify all these operations are equivalent in bit-level, and use the same uninterpreted functions to abstract them in the equivalence checking process.

Uninterpreted functions can handle overflow as well. For example, consider two computations $(A + B) - C$ and $(A - C) + B$. If we interpret the meaning of addition and substraction, the equivalence checker will report equivalence for these two computations. However, if $A+B$ overflows, whereas $A-C+B$ does not, the two computations will have different results. Uninterpreted functions do not consider the associativity, and therefore can correctly report the two are not equivalent.

3.4.2 Reducing False Negatives

Since the abstraction is conservative, the equivalence checker may falsely report two equivalent CCDFGs to be inequivalent. We need to have a way to reduce the chance of encountering false negatives. The basic idea is to use abstraction refinement technique to add more information into the model being verified. Currently, we support users to

manually decide the appropriate level of abstraction, and refine the CCDFG. There are several approaches to do the abstraction refinement:

- **Full interpretation:** If the operation can be specified in linear arithmetic or bit vector, we can use these theories and change the operations to be interpreted.

- **Partial interpretation:** Operations sometimes only need a partial interpretation instead of a full interpretation. For example, if we only care whether a multiplication operation will get 0 or not (SMT solver can not decide multiplication in general). We can partially interpreted the function as `mult(a,b): If a==0 OR b==0 THEN 0 ELSE mult'(a,b);`, where `mult'` is an uninterpreted function.

- **Bit-blasting:** We can bit-blasting some operations in bit-level. This may lose the advantage of SMT solver, since we only use the Boolean expression to represent the problem. But since SMT is not decidable in general, in the worst case, we have to bit-blasting everything into bit-level. In many cases, when the control flow does not tightly depend on the data, we may bit-blasting all the control conditions, while still take the advantage of uninterpreted functions on data.

3.5 EXPERIMENTAL RESULTS AND DISCUSSION

We implemented the above equivalence checking algorithm in the Intel *Forte* environment [21]. The equivalence checking is based on symbolic simulation; symbolic states are represented at bit-level using BDDs, and at word-level using theories supported by CVC3.

To assess the practical usefulness of my approach, I applied my equivalence checker to two designs: The GCD algorithm and a pipelined design synthesized by xPilot [10]. All experiments were conducted on a workstation with 3GHz Intel Xeon processor with 2GB memory.

	Before trans. seq. in Fig. 2.3				After trans. seq. in Fig. 2.3			
Bit Width	Circuit Nodes	# of Steps	Time (Sec.)	BDD Nodes	Circuit Nodes	# of Steps	Time (Sec.)	BDD Nodes
2	96	20	0.02	503	135	16	0.02	560
3	164	33	0.05	4772	240	22	0.04	5542
4	246	48	0.11	42831	373	34	0.11	55646
5	342	63	0.59	16244	534	40	0.79	90599
6	452	78	12.50	39968	723	58	17.78	51977
7	576	93	369.31	220891	940	70	376.48	84834
8	714	108	6850.56	1197604	1185	82	5798.03	589557

Table 3.1: Equivalence Checking Results for GCD in bit-level

We first checked the equivalence of the GCD algorithm totally in word-level, and using uninterpreted functions to represent modular operation. We finished the equivalence checking within one second for 8-bit GCD, and the equivalence checker reports the CCDFG and circuit is not equivalent. Since the control flow of GCD algorithm requires the full interpretation of modular operation, which is not supported by CVC3. In order to eliminate false negatives, we bit-blast all the operations in the CCDFG, and check the equivalence in bit-level. Table 3.1 shows the results for GCD before and after the transformation sequence in Fig. 2.3. In all cases, the equivalence checker can correctly report the equivalence of CCDFG and the synthesized circuit. However, the running time grows exponentially when the bit width increases. For the 8-bit GCD, the equivalence checking finished within 2 hours and the number of BDDs required is not prohibitive. It is interesting to note that the number of steps performed for equivalence checking provides a good estimation for the effectiveness of the transformation — the fewer number of steps needed to reach fixed-point, the more likely the circuit can run in less time. However, the performance gain in real circuit does not necessarily imply the performance improvement in equivalence checking, because the transformation

```
Pipeline(C,N) {
  result = 0;
  for (i = 0; i < N; i++) {
    tmp = i * C;
    result = result + tmp;
  }
  return result;
}
```

Figure 3.3: Behavioral description of a loop

also increases the number of circuit nodes, and the total number of modular operations required is the same as the unoptimized version. Therefore, increasing the circuit simulation time.

The other example we use is a for loop with and without pipelined design, generated by xPilot. Fig. 3.3 shows high-level design of the loop. Each of the operation "*" and "+" will take one cycle to finish. In the unpipelined design, the circuit takes two cycles for each iteration. The pipelined design only need one cycles for each iteration. We abstract both versions of the circuit as CCDFGs and check the equivalence with the CCDFG derived from Fig. 3.3. We use linear arithmetic to represent the addition and uninterpreted function to represent multiplication. Since the control flow does not need to check the result of the multiplication, my equivalence checker can successfully report the CCDFGs are equivalent for both unpipelined and pipelined design. The running time for the whole verification takes three minutes to finish for 8-bit implementations.

Chapter 4

CONCLUSION AND FUTURE WORK

We have demonstrated a formal sequential equivalence verification for high-level synthesis flow, based on symbolic simulation in both bit-level and word-level. By giving a mapping relation for operations extracted from high-level synthesis flows, we defined the notion of equivalence between a CCDFG and a circuit. The approach is capable of verifying circuits with complicated control logics and multiple clock cycles. The word-level representation scales the equivalence checking for many practical designs.

It must be admitted, however, that certification of practical synthesis flows is a substantial enterprise. Significant further research is necessary to facilitate the verification of a practical synthesis flow such as SPARK [14] or xPilot [10]. Further research is needed to improve the scalability, and accuracy of the equivalence checking. For scalability, research is needed to find ways to decompose a larger verification problem into smaller, more tractable ones. For example, by finding equivalent cut points between two CCDFGs, or by using the information provided by the theorem proving part to do induction or compositional model checking. An obvious way to improve accuracy is to develop an automatic counterexample guided abstraction refinement loop [7, 8]. For example, when the word-level model checker reports inequivalency, it generates an counterexample of the control path. We simulate this single control path in bit-level, and decide whether this is a spurious counterexample or not. If it is a spurious counterexample, we try to identify the cause of the imprecision and automatically add more information into the model. We need to carefully tune the level of abstraction, since too much abstraction yields inaccurate results, while too little results in complexity blow-up.

REFERENCES

[1] Cryptol: The Language of Cryptography. See URL: http://www.cryptol.net.

[2] Forte Design Systems. Behavioral Design Suite. See URL: http://www.forteds.com.

[3] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts.

[4] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen. Scalable sequential equivalence checking across arbitrary design transformations. In *International Conference on Computer Design*, 2006.

[5] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[6] Celoxica. DK design suite. See URL: http://www.celoxica.com.

[7] Y. Chen, Y. He, F. Xie, and J. Yang. Automatic abstraction refinement for generalized symbolic trajectory evaluation. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2007.

[8] Y. Chen, F. Xie, and J. Yang. Optimizing automatic abstraction refinement for generalized symbolic trajectory evaluation. In *Design Automation Conference (DAC)*, 2008.

[9] L. Claesen, M. Genoe, and E. Verlind. Implementation/specification verification by means of SFG-Tracing. In *CHARME*, 1993.

[10] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. Behavioral and Communication Co-Optimizations for Systems with Sequential Communication Media. In *DAC*, 2006.

[11] David W. Currie, Xiushan Feng, Masahiro Fujita, Alan J. Hu, Mark Kwan, and Sreeranga P. Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *International Journal of Parallel Programming*, 34(1):61–91, 2006.

[12] E. W. Dijkstra. Guarded Commands, Non-determinacy and a Calculus for Derivation of Programs. *Communications of the ACM*, 18:453–457, 1975.

[13] X. Feng, A. J. Hu, and J. Yang. Partitioned model checking from software specifications. In *ASP-DAC*, pages 583–587, 2005.

[14] D. Gajski, N. D. Dutt, A. Wu, and S. Lin. *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Norwell, MA, 1993.

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

[16] A. J. Hu. High-level vs. RTL combinational equivalence: An introduction. In *International Conference on Computer Design*, 2006.

[17] D. Kaiss, S. Goldenberg, Z. Hanna, and Z. Khasidashvili. Seqver: A sequential equivalence verifier for hardware designs. In *International Conference on Computer Design*, 2006.

[18] Alfred Kölbl, James H. Kukula, and Robert F. Damiano. Symbolic rtl simulation. In *Design Automation Conference (DAC)*, pages 47–52, 2001.

[19] Alfred Kölbl and Carl Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *International Journal of Parallel Programming*, 33(6):645–666, 2005.

[20] X. Leroy. Formal Certification of a Compiler back-end, or: Programming a Compiler with a Proof Assistant. In *POPL*, 2006.

[21] C.-J.H. Seger, R.B. Jones, J.W. O'Leary, T.F. Melham, M. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 24(9), 2005.

[22] J. Yang and C.-J. H. Seger. Generalized symbolic trajectory evaluation — abstraction in action. In *FMCAD*, November 2002.

[23] J. Yang and C.-J. H. Seger. Introduction to generalized symbolic trajectory evaluation. *Transaction on VLSI Systems*, 11(3), June 2003.