Formal Verification for High-Assurance Behavioral Synthesis*

Sandip Ray[1], Kecheng Hao[2], Yan Chen[3], Fei Xie[2], and Jin Yang[4]

[1] Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712
[2] Department of Computer Science, Portland State University, Portland, OR 97207
[3] Toyota Technological Institute at Chicago, Chicago, IL 60637†
[4] Strategic CAD Labs, Intel Corporation, Hillsboro, OR 97124

Abstract. We present a framework for certifying hardware designs generated through behavioral synthesis, by using formal verification to certify the associated synthesis transformations. We show how to decompose this certification into two components, which can be respectively handled by the complementary verification techniques, theorem proving and model checking. The approach produces a certified reference flow, composed of transformations distilled from production synthesis tools but represented as transformations on graphs with an associated formal semantics. This tool-independent abstraction disentangles our framework from the inner workings of specific synthesis tools while permitting certification of hardware designs generated from a broad class of behavioral descriptions. We provide experimental results suggesting the scalability on practical designs.

1 Introduction

Recent years have seen high complexity in hardware designs, making it challenging to develop reliable, high-quality systems through hand-crafted Register Transfer Level (RTL) or gate-level implementations. This has motivated a gradual migration away from RTL towards Electronic System Level (ESL) designs which permit description of design functionality abstractly in high-level languages, *e.g.*, SystemC. However, the ESL approach crucially depends on reliable tools for *behavioral synthesis*, that is, automated synthesis of a hardware circuit from its ESL description. Behavioral synthesis tools apply a sequence of transformations to compile the ESL description to an RTL design.

Several behavioral synthesis tools are available today [1–4]. Nevertheless, and despite its great need, behavioral synthesis has not yet found wide acceptance in industrial practice. A major barrier to its adoption is the lack of designers' confidence in correctness of synthesis tools themselves. The difference in abstraction level between a synthesized design and the ESL description puts the onus on behavioral synthesis to ensure that the synthesized design indeed conforms to the description. On the other hand, synthesis transformations necessary to produce designs satisfying the growing demands of performance and power include complex and aggressive optimizations which must respect subtle invariants. Consequently, synthesis tools are often either (a) error-prone or (b) overly conservative, producing circuits of poor quality and performance [4, 5].

† Yan Chen was a M.S. student at Portland State University when he participated in this research.

* This research was partially supported by a grant from Intel Corporation.

In this paper, we develop a scalable, mechanized framework for certifying behavioral synthesis flows. 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 the synthesized design is reduced to the question of analysis of the behavioral description. Our approach is distinguished by two key features:

– Our framework is *independent* of the inner workings of a specific tool, and can be applied to certify designs synthesized by different tools from a broad class of ESL descriptions. This makes our approach particularly suitable for certifying security-critical hardware which are often synthesized from domain-specific languages [6].
– The approach produces a certified *reference flow*, which makes explicit generic invariants that must be preserved by different transformations. The reference flow serves as a formal specification for reliable, aggressive synthesis transformations.

Formal verification has enjoyed significant successes in the analysis of industrial hardware designs [7, 8]. Nevertheless, applying formal verification directly to certify a *synthesized* design is undesirable for two reasons. First, it defeats the very purpose of behavioral synthesis as a vehicle for raising design abstraction since it requires reasoning at the level of the synthesized design rather than the behavioral description. Second, the cost of analyzing a complex design is substantial and the cost must be incurred for each design certification. Instead, our approach targets the *synthesis flow*, thereby raising the level of abstraction necessary for design certification.

In the remainder of this section, we first provide a brief overview of behavioral synthesis with an illustrative example; we then describe our approach in greater detail.

1.1 Behavioral Synthesis and An Illustrative Example

A behavioral synthesis tool accepts a design description and a library of hardware resources; it performs a sequence of transformations on the description to generate RTL. The transformations are roughly partitioned into the following three phases.

– **Compiler transformations.** These include loop unrolling, common subexpression elimination, copy propagation, code motion, etc. Furthermore, expensive operations (*e.g.*, division) are often replaced with simpler ones (*e.g.*, subtraction).
– **Scheduling.** This phase determines the clock step for each operation. The ordering between operations is constrained by the data and control dependencies. Scheduling transformations include chaining operations across conditional blocks and decomposing one operation into a sequence of multi-cycle operations based on resource constraints. Furthermore, several compiler transformations are employed, exploiting (and creating opportunities for) operation decomposition and code motions.
– **Resource binding and control synthesis.** This phase binds operations to functional units, allocates and binds registers, and generates the control circuit to implement the schedule.

After these transformations, the design can be expressed as RTL. This design is subjected to further manual optimizations to fine-tune for performance and power.

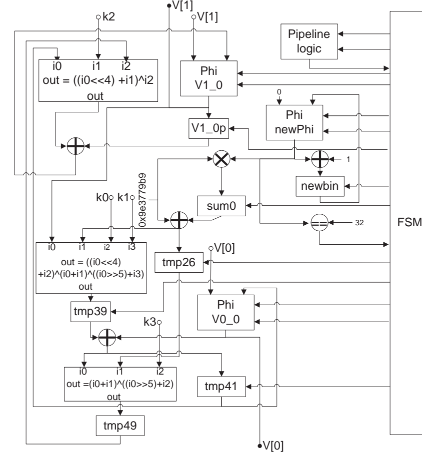
Each synthesis transformation is non-trivial. The consequence of their composition is a significant difference in abstraction from the original description. To illustrate this,

```
void encrypt (uint32_t* v, uint32_t* k) {
  /* set up */
  uint32_t v0=v[0], v1=v[1], sum=0, i;
  /* a key schedule constant */
  uint32_t delta=0x9e3779b9;
  /* cache key */
  uint32_t k0=k[0], k1=k[1],
           k2=k[2], k3=k[3];

  /* basic cycle start */
  for (i=0; i < 32; i++) {
    sum += delta;
    v0 += ((v1<<4)+k0)^(v1 + sum)
          ^((v1>>5)+k1);
    v1 += ((v0<<4)+k2)^(v0 + sum)
          ^((v0>>5)+k3);
  }

  /* end cycle */
  v[0]=v0; v[1]=v1;
}
```

(A)



(B)

Fig. 1. (A) C code for TEA encryption function. (B) Schema of RTL synthesized by AutoPilot.

consider the synthesis of the Tiny Encryption Algorithm (TEA) [9]. Fig. 1 shows a C implementation and the circuit synthesized by the AutoPilot behavioral synthesis tool [10]. The following transformations are involved in the synthesis of the circuit.

- In the first phase, constant propagation removes unnecessary variables.
- In the second phase, the key scheduling transformation performed is *pipelining*, to enable overlapping execution of operations from different loop iterations.
- In the third phase, operations are bound to hardware resources (*e.g.*, "+" operation to an adder), and the FSM module is generated to schedule circuit operations.

Each transformation must respect subtle design invariants. For instance, paralleling operations from different loop iterations must avoid race conditions, and scheduling must respect data dependencies. Since such considerations are entangled with low-level heuristics, it is easy to have errors in the synthesis tool implementation, resulting in buggy designs [5]. However, the difference in abstraction level makes direct comparison between the C and RTL descriptions impractical; performing such comparison through sequential equivalence checking [11] requires cost-prohibitive symbolic co-simulation to check input/output correspondence.

1.2 Approach Overview

We address the above issue by breaking the certification of behavioral synthesis transformations into two components, ***verified*** and ***verifying***.[5] Fig. 2 illustrates our framework. A ***verified*** transformation is formally certified once and for all using theorem

[5] The terms "***verified***" and "***verifying***" as used here are borrowed from analogous notions in the compiler certification literature.

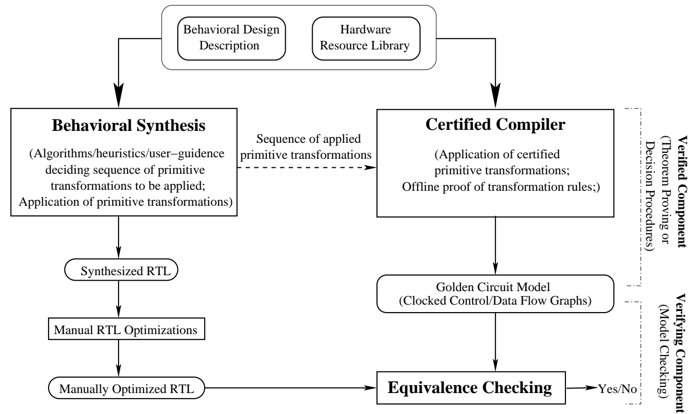


Fig. 2. Framework for certification of behavioral synthesis flows

proving; a ***verifying*** transformation is not itself verified, but each instance is accompanied by a verification of correspondence between input and output. The viability of decomposition is justified by the nature of behavioral synthesis. Transformations applied at the higher level, (*e.g.*, compiler and scheduling transformations) are generic. The cost of a monolithic proof is therefore mitigated by the reusability of the transformation over different designs. Such transformations make up the ***verified*** component. On the other hand, the optimizations performed at the lower levels are unique to the design being synthesized; these transformations constitute the ***verifying*** component. Since the verification is discharged per instance, it must be fully automatic. However, these transformations tend to be localized and independent of global invariants, making it tractable to verify them automatically by sequential equivalence checking.

1.3 Golden Circuit Model and Synthesis Certification

In a practical synthesis tool, transformations are implemented with low-level, optimized code. A naive approach for the ***verified*** component, *e.g.*, to formally verify such a tool with all optimizations would be prohibitive. Furthermore, such an approach would tie the framework to a single tool, limiting reusability.

To mitigate this challenge, we develop a formal, graph-based abstraction called *clocked control/data flow graph* (CCDFG), which serves as the universal golden circuit model. We discuss our formalization of CCDFG in Section 2. CCDFG is an abstraction of the control/data flow graph (CDFG) — used as an intermediate representation in most synthesis tools — augmented with a schedule. The close connection between the formal abstraction and the representation used in a synthesis flow enables us to view synthesis transformations as transformations on CCDFG, while obviating a morass of tool-specific details. We construct a *reference flow* as a sequence of CCDFG transformations as follows: each transformation generates a CCDFG that is guaranteed to preserve semantic correspondence with its input. A production transformation is decomposed into *primitive transformations*, together with algorithms/heuristics that determine the

application sequence of these transformations. Once the primitive transformations are certified, the algorithms or heuristics do not affect the correctness of a transformation sequence, only the performance. The reference flow requires no knowledge about the algorithms/heuristics which are often confidential to a synthesis tool.

Given a synthesized hardware design $\mathcal{D}$ and its corresponding behavioral description, the certification of the hardware can be mechanically performed as follows.

- Extract the CCDFG $\mathcal{C}$ from the behavioral description.
- Apply the certified primitive transformations from the reference flow, following the application sequence provided by the synthesis tool. The result is a CCDFG $\mathcal{C}'$ that is close to to $\mathcal{D}$ in abstraction level.
- Apply equivalence checking to guarantee correspondence between $\mathcal{C}'$ and $\mathcal{D}$.

The overall correctness of this certification is justified by the correctness of the ***verified*** and ***verifying*** components and their coupling through the CCDFG $\mathcal{C}'$.

How does the approach disentangle the certification of a synthesized hardware from the inner workings of the synthesis tool? Although each certified transformation mimics a corresponding transformation applied by the tool, from the perspective of *certifying* the hardware they are merely heuristic guides transforming CCDFGs to facilitate equivalence checking: certification of the synthesized hardware reduces to checking that the *initial* CCDFG reflects the design intent. The initial CCDFG can be automatically extracted from the synthesis tools' initial internal representation.[6] Furthermore, the framework abstracts low-level optimizations making the verification problem tractable.

The rest of the paper is organized as follows. In Section 2 we present the semantics of CCDFG. In Section 3 we discuss how to use theorem proving to verify the correctness of generic CCDFG transformations. In Section 4 we present our equivalence checking procedure. We provide initial experimental results in Section 5, discuss related work in Section 6, and conclude in Section 7.

2 Clocked Control/Data Flow Graphs

A CCDFG can be viewed as a formal *control/data flow graph* (CDFG) — used as internal representation in most synthesis tools including Spark and Autopilot — augmented with a schedule. Fig. 3 shows two CCDFGs for the TEA encryption. The semantics of CCDFG are formalized in the logic of the ACL2 theorem prover [12]. This section briefly discusses the formulation of a CCDFG; for a more complete account, see [13].

The formalization of CCDFG assumes that the underlying language provides the semantics for a collection *ops* of *primitive operations*. The primitive operations in Fig. 1 include comparison and arithmetic operations. We also assume a partition of design variables into *state variables* and *input variables*. Variable assignments are assumed to be in a Static in Single Static Assignment (SSA) form. Design descriptions are assumed to be amenable to control and data flow analysis. Control flow is broken up into basic

[6] Since the input description is normally unclocked, the initial CCDFG does not contain schedule information, and can be viewed as a CDFG. Schedules are generated by synthesis transformations that turn the unclocked representation to a clocked one.

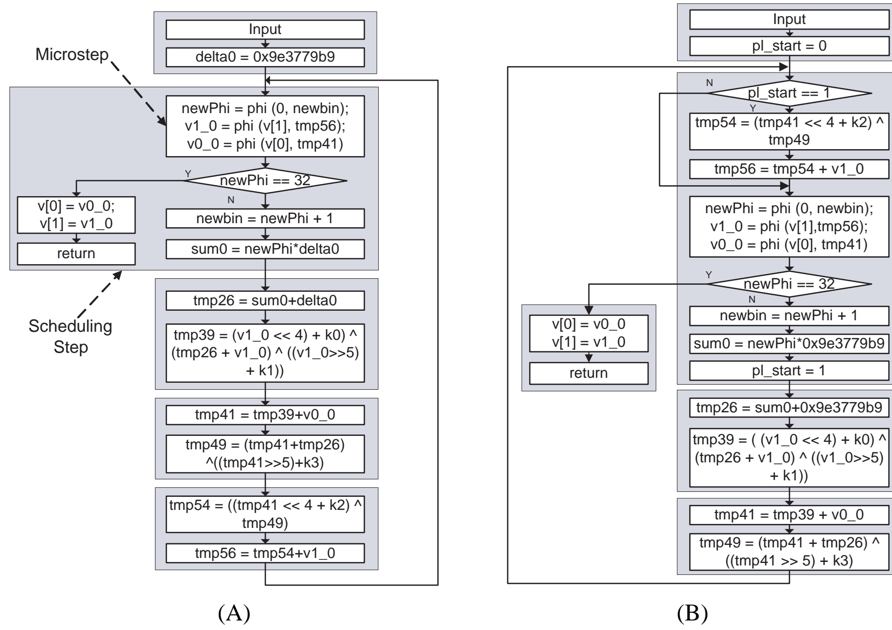


Fig. 3. (A) Initial CCDFG of TEA encryption function. (B) Transformed CCDFG after pipelining. The shaded regions represent scheduling steps, and white boxes represent microsteps. For brevity, only the control flow is shown; data flow is omitted. Although the underlying operations are assumed to be in SSA form, the diagrams aggregate several single assignments for simplicity.

blocks. Data dependency is given by "read after write" paradigm: op_j is data dependent on 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 1 (Control 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. A* data flow graph G_D *over ops is a directed acyclic graph with vertex set ops. A* control flow graph G_C *is a graph with vertex set bb and each edge labeled with an assertion over* V*.*

An edge in G_D from op_i to op_j represents data dependency, 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 of. An assertion on an edge holds whenever program control makes the corresponding transition.

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

Definition 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 that $op_a \prec_k op_b$ 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 $op_a \not\prec_k op_b$ 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.*

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. Note that we treat different instances of the same operation as different (with same semantics); this permits stipulation of H as a function instead of a relation, and simplifies the formalization. In Fig. 3, each white box corresponds to a microstep partition. Since G_D is acyclic, $\prec_k$ is an irreflexive partial order on $ops(bb_k)$ and the notion of microstep partition 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.

CCDFGs are formalized 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 they belong to the same scheduling step. Informally, if two microsteps in M are in the same group in T then they are executed within the same clock cycle.

Definition 4 (CCDFG). *A* CCDFG *is a tuple $G \triangleq \langle G_{CD}, M, T\rangle$, where G_{CD} is a CDFG, M is a microstep partition of G_{CD}, and T is a schedule of M.*

We formalize CCDFG executions through a state-based semantics. A *CCDFG state* is a valuation of state variables, and a *CCDFG input* is a valuation of input variables. We also assume a well-defined *initial state*. Given a sequence $\mathcal{I}$ of inputs, an *execution* of a CCDFG $G = \langle G_{CD}, M, T\rangle$ is a sequence of CCDFG states that corresponds to an evaluation of the microsteps in M respecting T.

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. To formalize this in ACL2's first order logic, the output is restricted to a Boolean expression of the state variables; the domain of each state variable itself is unrestricted, which enables us to represent programs such as the Greatest Common Divider (GCD) algorithm that do not return Boolean values. 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.

3 Certified Compilation

Certifying a transformation $\mathcal{T}$ requires showing that if the application of $\mathcal{T}$ on a CCDFG G generates a new CCDFG G', then there is provable correspondence between the executions of G and G'. The certification process crucially depends on a formal notion of correspondence to relate the executions of G and G'. Note that the notion must comprehend differences between execution order of operations as long as the sequence of observations is unaffected. The notion we use is loosely based on *stuttering trace containment* [14, 15]. Roughly, a CCDFG G' *refines* G if for each execution of G' there is an execution of G that produces the same observable behavior up to stuttering. We formalize this notion below.

Definition 5 (Compressed Execution). *Let $\mathcal{E} \triangleq s_0, s_1, \ldots$ be an execution of CCDFG G and F be a set of output functions over G. The* compression *of $\mathcal{E}$ under F is the subsequence of $\mathcal{E}$ obtained by removing each s_i such that $f(s_i) = f(s_{i+1})$ for every $f \in F$.*

Definition 6 (Trace Equivalence). *Let G and G' be two CCDFGs on the same set of state and input variables, $\mathcal{E}$ and $\mathcal{E}'$ be executions of G and G' respectively, and F be a set of output functions. We say that $\mathcal{E}$ is* trace equivalent *to $\mathcal{E}'$ if the observable behavior of the compression of $\mathcal{E}$ under F is the same as the observable behavior of the compression of $\mathcal{E}'$ under F.*

Definition 7 (CCDFG Refinement). *We say that a CCDFG G'* refines *G if for each execution $\mathcal{E}'$ of G' there is an execution $\mathcal{E}$ of G such that $\mathcal{E}$ is trace equivalent to $\mathcal{E}'$.*

Remark 1. For the ***verified*** component, we use refinement instead of full equivalence as a notion of correspondence between CCDFGs, to permit connecting the same ESL description with a number of different concrete implementations. In the ***verifying*** framework, we will use a stronger notion of equivalence (and indeed, equivalence without stuttering), to facilitate sequential equivalence checking.

In addition to showing that a transformation on a CCDFG G produces a refinement of G, we must 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 to facilitate a subsequent transformation. To make explicit the notion of applicability of a transformation, we view a transformation as a "guarded command" $\tau \triangleq \langle pre, \mathcal{T}, post \rangle$: τ is applicable to a CCDFG which satisfies *pre* and produces a CCDFG which satisfies *post*.

Definition 8 (Transformation Correctness). *A transformation $\tau \triangleq \langle pre, \mathcal{T}, post \rangle$ is* correct *if the result of applying $\mathcal{T}$ to any CCDFG G satisfying pre refines G and satisfies $post$.*

The following theorem is trivial by induction on the sequence of transformations. Here $[\mathcal{T}_0, \ldots, \mathcal{T}_n]$ represents the composition of $\mathcal{T}_0, \ldots, \mathcal{T}_n$.

Theorem 1 (Correctness of Transformation Sequences). *Let $\tau_0, \ldots, \tau_n$ be some sequence of correct transformations, where $\tau_i \triangleq \langle pre_i, \mathcal{T}_i, post_i \rangle$, Let $post_i \Rightarrow pre_{i+1}$, $1 \leq i < n$. Then the transformation $\langle pre_1, [\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_n], post_n \rangle$ is correct.*

Theorem 1 justifies decomposition of a transformation into a sequence of primitive transformations. Note that the proof of Theorem 1 is independent of a specific transformation. We thus construct a *reference flow* as follows. (1) Identify and distill a sequence $\tau_0, \ldots, \tau_n$ of primitive transformations; (2) verify τ_i individually; and (3) check that for each $0 \leq i < n$, $post_i \Rightarrow pre_{i+1}$. Theorem 1 guarantees the correctness of the flow.

Verifying the correctness of individual guarded transformations using theorem proving might involve significant manual effort. To ameliorate this cost, we identify and derive *generic theorems* that can certify a class of similar transformations. As a simple example, consider any transformation that refines the schedule. The following theorem states that each such transformation is correct.

Theorem 2 (Correctness of Schedule Refinement). *Let $G \triangleq \langle G_{CD}, M, T \rangle$ and $G' \triangleq \langle G_{CD}, M, T' \rangle$ be CCDFGs such that for any two microsteps $m_i, m_j \in M$ if T' assigns m_i and m_j the same group then so does T. Then G' is a refinement of G.*

Theorem 2 is admittedly trivial; it is only shown here for illustration purposes. However, the same approach can verify more complex transformations. For example, consider the constant propagation and pipelining transformations shown in Figure 3 for our TEA example. The implementations of these transformations involve significant heuristics, for instance, to determine whether to apply the transformations in a specific case, how many iterations of the loop should be pipelined, etc. However, from the perspective of correctness, the only relevant conditions about the two transformations are: (1) if a variable v is assigned a constant c, then v can be eliminated by replacing each occurrence with c; and (2) a microstep m_i can be overlapped with microstep m_j from a subsequent iteration if for each $op_i \in m_i$ and $op_j \in m_j$, $op_j \not\prec op_i$ in G. Since these conditions are independent of a specific design (*e.g.*, TEA) to which the transformation is applied, the same certification can be used to justify its applicability for diverse designs. The approach is viable because we employ theorem proving which supports an expressive logic, thereby permitting stipulation of the general conditions above as formal predicates in the logic. For example, as we show in previous work [16], we can make use of first-order quantification to formalize a generic refinement proof of arbitrary pipelines, which is directly reusable for verification of the pipeline transformation in our framework. Another generic transformation that is widely employed in behavioral synthesis is *operation balancing*; its correctness depends only on the fact that the operations involved are associative and commutative and can be proven for CCDFGs containing arbitrary associative-commutative operations.

We end the discussion of the ***verified*** framework with another observation. Since the logic of ACL2 is executable, *pre* and *post* can be efficiently executed for a given concrete transformation. Thus, a transformation $\tau \triangleq \langle pre, \mathcal{T}, post \rangle$ can be applied *even before verification* by using *pre* and *post* for runtime checks: if a CCDFG G indeed satisfies *pre* and the application of τ on G results in a CCDFG satisfying *post* then the *instance* of application of τ on G can be composed with other compiler transformations; furthermore, the expense of the runtime assertion checking can be alleviated by generating a *proof obligation for a specific instance*, which is normally more tractable than a monolithic generic proof of the correctness of τ. This provides a trade-off between the computational expense of runtime checks and verification of individual instances with a (perhaps deep) one-time proof of the correctness of a transformation.

4 Equivalence Checking

We now discuss how to check equivalence between a CCDFG and its synthesized circuit. The ***verified*** component facilitates close correspondence between the transformed CCDFG and the synthesized circuit, critical to the scalability of equivalence checking.

4.1 Circuit Model

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

Definition 9 (Circuit). *A circuit is a tuple $M = \langle I, N, F \rangle$ where I is a vector of inputs; 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 an expression over $N_c \cup N_d \cup I$ and for each latch $d \in N_d$, F_d maps each latch d to $n \in N_c \cup N_d \cup I$ 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. Given a sequence of valuations to the inputs $i_0, i_1, \ldots$, 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}. The *observable behavior* of the circuit is the sequence of valuations of the *outputs* which are a subset of latches and combinational nodes.

4.2 Correspondence between CCDFGs and Circuits

Given a CCDFG G and a synthesized circuit M, it is tempting to define a notion of correspondence as follows: (1) Establish a fixed mapping between the state variables of G and the latches in M, and (2) stipulate an execution of G to be equivalent to an execution of M if they have the same observable behavior. However, this does not work in general since the mappings between state variables and latches may be different in each clock cycle. To address this, we introduce $EMap : ops \rightarrow N_c$, mapping CCDFG *operations* to the combinational nodes in the circuit: each operation is mapped to the combinational node that implements the operation; the mapping is independent of clock cycles. Fig. 4 shows the mapping for the synthesized circuit of TEA. Recall from Section 1.1 that the FSM decides the *control signals* for the circuit; the FSM is thus excluded from the mapping. We now define the equivalence between G and M.

Definition 10. *A CCDFG state x of G is equivalent to a circuit state s of M with respect to an input i and a microstep partition t, if for each operation op in t, the inputs to op according to x and i are equivalent to the inputs to $EMap(op)$ according to s and $EMap(i)$, i.e., the values of each input to op and the corresponding input to $EMap(op)$ are equivalent, and the outputs of op are equivalent to the outputs of $EMap(op)$.*

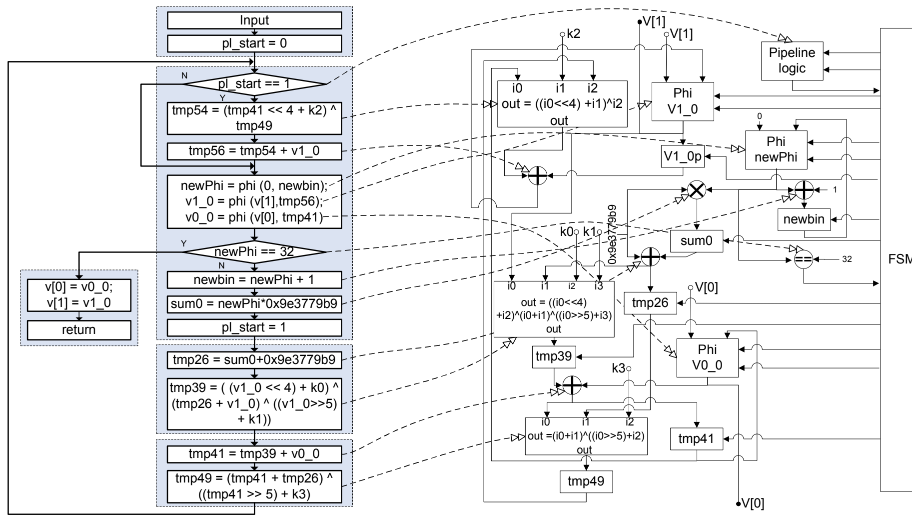


Fig. 4. Synthesized circuit for TEA and the corresponding operation mapping with pipelined CCDFG; dotted lines represent mapping from CCDFG operations to combinational circuit nodes.

Definition 11. *Given a CCDFG G and a circuit M, G is equivalent to M if and only if for any execution $[x_0, x_1, x_2, \ldots]$ of G generated by an input sequence $[i_0, i_1, i_2, \ldots]$ and by microstep partition $[t_0, t_1, \ldots]$ of G, and the state sequence $[s_0, s_1, s_2, \ldots]$ of M generated by the input sequence $[EMap(i_0), EMap(i_1), EMap(i_2), \ldots]$, x_k and s_k are equivalent with respect to t_k under i_k, $k \geq 0$.*

4.3 Dual-Rail Simulation for Equivalence Checking

We check equivalence between CCDFG G and circuit M by dual-rail symbolic simulation (Fig. 5); the two rails simulate G and M respectively, and are synchronized by clock cycle. The equivalence checking in clock cycle k is conducted as follows:

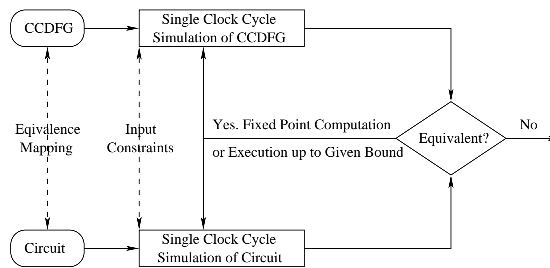


Fig. 5. Dual-Rail simulation scheme for equivalence checking between CCDFG and circuit.

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 $EMap(op)$. If yes, continue; otherwise, report inequivalence.
2. G is simulated by executing t_k on x_k under i_k to compute x_{k+1} and recording the outputs of each $op \in t_k$. M is simulated for one clock cycle from s_k under input $EMap(i_k)$ to compute s_{k+1}. The outputs for each op are checked for equivalence with the outputs of $EMap(op)$. If yes, continue; otherwise, report inequivalence.
3. The next scheduling step t_{k+1} is determined from control flow. If t_k has multiple outgoing control edges, the last microstep of t_k executed is identified. The outgoing control edge from this microstep whose condition evaluates to true leads to t_{k+1}.

We permit both bounded and unbounded (fixed-point) simulations. In particular, the simulation proceeds until (i) the equivalence check fails, (ii) the end of a bounded input sequence is reached, or (iii) a fixed point is reached for an unbounded input sequence.

We have implemented the dual-rail scheme on the bit level in the Intel *Forte* environment [17], where symbolic states are represented using BDDs. We have also implemented the scheme on the word level with several built-in optimizations, using Satisfiability Modulo Theories (SMT); this is viable since word-level mappings between operations and circuit nodes are explicit. We use bit-vectors to encode the variables in the CCDFG and the circuit; the SMT engine checks input/output equivalence and determines control paths. Our word-level checker is based on the CVC3 SMT engine [18].

The bit-level and word-level checkers are complementary. The bit-level checker ensures that the equivalence checking is decidable, while the word-level checker provides the optimizations crucial to scalability. The word-level checker can make effective use of results from bit-level checking in many cases. One typical scenario is as follows. Suppose M is a design module of modest complexity but is awkward to check at word-level. Then the bit-level checker is used to check the equivalence of the CCDFG of M with its circuit implementation; when the word-level checker is used for equivalence checking of a module that calls M, it skips the check of M, treating the CCDFG of M and its circuit implementation as equivalent black boxes.

5 Experimental Results

We used the bit-level checker on a set of CCDFGs for GCD and the corresponding circuits synthesized by AutoPilot. The experiments were conducted on a workstation with 3GHz Intel Xeon processor with 2GB memory. Table 1 shows the statistics before and after schedule refinement (Theorem 2). Since we bit-blast all CCDFG operations, the running time grows exponentially with the bit width; for 8-bit GCD, checking requires about 2 hours. It is interesting to understand how schedule refinement affects the performance of equivalence checking. Schedule refinement partitions operations in the loop body into two clock cycles. This does not change fixed-point computation; however, the number of cycles for which the circuit is simulated doubles. For small bit-widths, the running time after schedule refinement is about two times slower than that before. However, for large bit widths, the running time is dominated by the complexity of the CCDFG simulation instead of the circuit simulation. The decrease in time with the increase in bit width from 7 to 8 is likely due to BDD variable reordering.

Table 1. Bit-level equivalence checking statistics

Circuit		Before schedule refinement		After schedule refinement	
Bit Width	# of Nodes	Time (Sec.)	BDD Nodes	Time (Sec.)	BDD Nodes
2	96	0.02	503	0.02	783
3	164	0.05	4772	0.07	11113
4	246	0.11	42831	0.24	20937
5	342	0.59	16244	1.93	99723
6	452	12.50	39968	27.27	118346
7	576	369.31	220891	383.98	164613
8	714	6850.56	1197604	3471.74	581655

Using our word-level checker, we have checked several RTL designs synthesized by AutoPilot with CCDFGs derived from AutoPilot's intermediate representations; the statistics are shown in Table 2. The designs illustrate different facets of the framework.

Table 2. Word-level equivalence checking statistics

Design	GCD	TEA	DCT	3DES	3DES_key
C Code Size (# of Lines)	14	12	52	325	412
RTL Size (# of Lines)	364	1001	688	18053	79976
Time (Seconds)	2	15.6	30.1	355.7	2351.7
Memory (Megabytes)	4.1	24.6	49.2	59.4	307.2

GCD contains a loop whose number of iterations depends on the inputs. TEA has an explicitly bounded loop. DCT contains sequential computation without loop. 3DES represents a practical design of significant size. 3DES_key is included to illustrate the scalability of our approach on relatively large synthesized designs. The results demonstrate the efficacy of our word-level equivalence checking. In contrast, full word-level symbolic simulation comparing the input/output relations of C and RTL runs out of memory on all the designs but DCT (for which it needs twice as much time and memory).

6 Related Work

An early effort [19] on verification of high-level synthesis targets the behavioral portion of VHDL [20]. A translation from behavioral VHDL to dependence flow graphs [21] was verified by structural induction based on the CSP [22] semantics. Recently, there has been research on certified synthesis of hardware from formal languages such as HOL [23] in which a compiler that automatically translates recursive function definitions in HOL to clocked synchronous hardware has been developed. A certified hardware synthesis from programs in Esterel, a synchronous design language, has been also been developed [24] in which a variant of Esterel was embedded in HOL.

Dave [25] provides a comprehensive bibliography of compiler verification. One of the earliest work on compiler verification was the Piton project [26], which verified a

simple assembly language compiler. Compiler certification forms a critical component of the Verisoft project [27], aiming at correctness of implementations of computing systems with both hardware and software components. The Verifix [28] and CompCert [29] projects have explored a general framework for certification of compilers for various C subsets [30, 31]. There has also been work on a ***verifying*** compiler, where each instance of a transformation generates a proof obligation discharged by a theorem prover [32].

There has been much research on sequential equivalence checking (SEC) between RTL and gate-level hardware designs [33, 34]. Research has also be done on combinational equivalence checking between high-level designs in software-like languages (*e.g.*, SystemC) and RTL-level designs [11]. There has also been effort for SEC between software specifications and hardware implementations [35]: GSTE assertion graphs [36] were extended so that an assertion graph edge have pre and post condition labels, and also associated assignments that update state variables. There has also been work on equivalence checking with other graph representations, *e.g.*, Signal Flow Graph [37].

7 Conclusion

We have presented a framework for certifying behavioral synthesis flows. The framework includes a combination of ***verified*** and ***verifying*** paradigms: high-level transformations are certified once and for all by theorem proving, while low-level tweaks and optimizations can be handled through model checking. We demonstrated the use of the CCDFG structure as an interface between the two components. Certification of different compiler transformations is uniformly specified by viewing them as manipulation of CCDFGs. The transformed CCDFG can then be used for equivalence checking with the synthesized design. One key benefit of the approach is that it obviates the need for developing formal semantics for each different intermediate representation generated by the compiler. Furthermore, the low-level optimizations implemented in a synthesis tool are abstracted from the reasoning framework without weakening the formal guarantee on the synthesized design. Our experimental results indicate that the approach scales to verification of realistic designs synthesized by production synthesis tools.

In future work, we will make further improvements to improve scalability. In the ***verified*** component, we are formalizing other generic transformations *e.g.*, code motion across loop iterations. In the ***verifying*** component, we are considering the use of theorem proving to partition a CCDFG into smaller subgraphs for compositional certification. We are also exploring ways to tolerate limited perturbations in mappings between CCDFGs and circuits (*e.g.*, due to manual RTL tweaks) in their equivalence checking.

References

1. Forte Design Systems: Behavioral Design Suite. http://www.forteds.com.
2. Celoxica: DK Design Suite. http://www.celoxica.com.
3. Cong, J., Fan, Y., Han, G., Jiang, W., Zhang, Z.: Behavioral and Communication Co-Optimizations for Systems with Sequential Communication Media. In: DAC. (2006)
4. Gajski, D., Dutt, N.D., Wu, A., Lin, S.: High Level Synthesis: Introduction to Chip and System Design. Kluwer Academic Publishers (1993)

5. Kundu, S., Lerner, S., Gupta, R.: Validating High-Level Synthesis. In: CAV. (2008)
6. Galois, Inc.: Cryptol: The Language of Cryptography (2007)
7. Russinoff, D.: A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD-K7 Floating-point Multiplication, Division, and Square Root Instructions. JCM **1** (1998)
8. O'Leary, J., Zhao, X., Gerth, R., Seger, C.J.H.: Formally Verifying IEEE Compliance of Floating-point Hardware. Intel Technology Journal **Q1** (1999)
9. Wheeler, D.J., Needham, R.M.: Tea, a tiny encryption algorithm. In: Fast Software Encryption. (1994)
10. AutoESL: AutoPilot Reference Manual. AutoESL. (2008)
11. Hu, A.J.: High-level vs. RTL combinational equivalence: An introduction. In: ICCD. (2006)
12. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Boston, MA (June 2000)
13. Ray, S., Chen, Y., Xie, F., Yang, J.: Combining theorem proving and model checking for certification of behavioral synthesis flows. Technical Report TR-08-48, University of Texas at Austin (2008)
14. Abadi, M., Lamport, L.: The Existence of Refinement Mappings. TCS **82**(2) (1991)
15. Lamport, L.: What Good is Temporal Logic? Information Processing **83** (1983)
16. Ray, S., Hunt, Jr., W.A.: Deductive Verification of Pipelined Machines Using First-Order Quantification. In: CAV. (2004)
17. Seger, C.J.H., Jones, R., O'Leary, J., Melham, T., Aagaard, M., Barrett, C., Syme, D.: An industrially effective environment for formal hardware verification. TCAD **24**(9) (2005)
18. Barrett, C., Tinelli, C.: CVC3. In: CAV. (2007)
19. Chapman, R.O.: Verified high-level synthesis. PhD thesis, Ithaca, NY, USA (1994)
20. IEEE: IEEE Std 1076: IEEE standards VHDL language reference manual
21. Johnson, R., Pingali, K.: Dependence-based program analysis. In: PLDI. (1993)
22. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
23. Gordon, M., Iyoda, J., Owens, S., Slind, K.: Automatic formal synthesis of hardware from higher order logic. TCS **145** (2006)
24. Schneider, K.: A verified hardware synthesis for Esterel. In: DIPES. (2000)
25. Dave, M.A.: Compiler verification: a bibliography. SIGSOFT SEN **28**(6) (2003)
26. Moore, J.S.: Piton: A Mechanically Verified Assembly Language. Kluwer Academic Publishers (1996)
27. Verisoft Project: http://www.verisoft.de.
28. Verifix Project: http://www.info.uni-karlsruhe.de/˜verifix.
29. CompCert Project: http://pauillac.inria.fr/˜xleroy/compcert.
30. Leinenbach, D., Paul, W.J., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In: SEFM. (2005)
31. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL. (2006)
32. Pike, L., Shields, M., Matthews, J.: A Verifying Core for a Cryptographic Language Compiler. In: ACL2. (2006)
33. Baumgartner, J., Mony, H., Paruthi, V., Kanzelman, R., Janssen, G.: Scalable sequential equivalence checking across arbitrary design transformations. In: ICCD. (2006)
34. Kaiss, D., Goldenberg, S., Hanna, Z., Khasidashvili, Z.: Seqver: A sequential equivalence verifier for hardware designs. In: ICCD. (2006)
35. Feng, X., Hu, A.J., Yang, J.: Partitioned model checking from software specifications. In: ASP-DAC. (2005)
36. Yang, J., Seger, C.J.H.: Introduction to generalized symbolic trajectory evaluation. TVLSI **11**(3) (2003)
37. Claesen, L., Genoe, M., Verlind, E.: Implementation/specification verification by means of SFG-Tracing. In: CHARME. (1993)