Combining Theorem Proving and Model Checking for Certification of Behavioral Synthesis Flows

Sandip Ray
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712
sandip@cs.utexas.edu

Yan Chen and Fei Xie
Department of Computer Science
Portland State University
Portland, OR 97207
{chenyan,xie}@cs.pdx.edu

Jin Yang
Strategic CAD Labs, DTS
Intel Corporation
Hillsboro, OR 97124
jin.yang@intel.com

***Abstract*—We develop a framework for certifying behavioral synthesis flows. Certification 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. We discuss how CCDFGs facilitate both theorem proving and model checking. The semantics of CCDFGs have been formalized with the ACL2 theorem prover, and the formalization used to certify generic synthesis transformations. Finally, we extend GSTE to model check synthesized netlists with respect to CCDFG specifications.**

I. 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. Behavioral synthesis [1], [2], [3], [4] 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.

In spite of its promise, behavioral synthesis has not yet found wide acceptance in engineering practice [4]. 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.

The goal of our research is to 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 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 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.

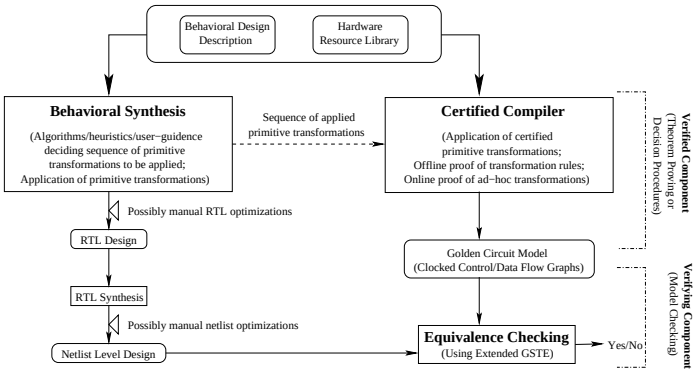

Fig. 1. Framework for Analysis of Behavioral Synthesis Flow

Our overall analysis framework is shown in Fig. 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 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.

The framework requires a smooth interface between the ***verified*** and ***verifying*** components. In our approach, this

[1]The terms "***verified***" and "***verifying***" as used here have been borrowed from analogous notions in the compiler certification literature [5], [6].

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, we have 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 netlist. Our equivalence checking paradigm leverages the high capacity of GSTE-style model checking [7] 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 netlist implementation. The two rails are synchronized by clock cycle.

The remainder of the paper is organized as follows. In Section II we present the formal semantics of CCDFG. In Section III we develop the formal notion of correspondence between CCDFG transformations, and discuss how to use theorem proving to verify the correctness of transformations with respect to this notion. In Section IV we present our equivalence checking procedure and discuss its scalability. We discuss related work in Section V, and conclude in Section VI.

II. Semantics of CCDFG

In this section, we formulate 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. Although the definitions presented here have been formalized in the ACL2 theorem prover, we adhere to traditional mathematical description in this presentation.

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 II.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 II.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.

In ACL2, we represent sets as finite lists and the mapping H as an association list. A graph is represented by an association list mapping each node to its neighbors. The syntactic structure is formalized by a predicate which checks that its argument corresponds to the representation of a CDFG.

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 II.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 $ops(bb_k) \triangleq \{op_i : H(op_i) = bb_k\}$ 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 $op_a \neq op_b$ 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 IV-A), 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. In ACL2, this is formalized through encapsulation [8], which allows introduction of functions with constraints rather than full definitions; the effect of executing the operations is represented as a constrained function, but the constraints include the semantics of assignment, comparison, and swap instructions. 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 II.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.

Definition II.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.

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

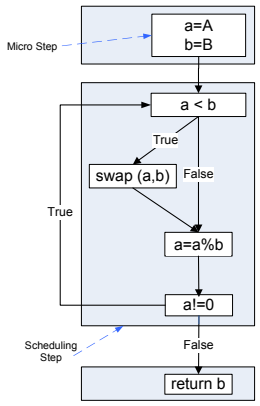

Fig. 2. Source code for GCD and the corresponding CCDFG. In the CCDFG, each while box denotes a micro step and each shaded region denotes a scheduling step. The primitive operations here are assignment, comparison, modular division, and swap. To simplify presentation, only control dependency edges are shown and data dependency edges are omitted.

Fig. 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 II.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 II.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 II.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 under-

lying 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 netlist 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 GCD example 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.

III. Certified Compilation

We now discuss the ***verified*** component of the framework, namely certification of transformations using theorem proving; in Section IV we will consider the ***verifying*** component.

The central element of the ***verified*** component is the definition of the notion of correspondence used to relate the input and output transformations. Note that for scalability, the notion must be reusable over compiler transformations which operate on designs at different levels of abstraction. We achieve this through a notion of correspondence loosely based on *stuttering trace containment* [9], [10].[2] The notion is inspired by work on well-founded bisimulations (WEBs) [11], [12] in ACL2 proofs of correctness of reactive systems. 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 III.1 (Compressed Execution)**.** Let $\mathcal{E} \triangleq s_0, s_1, \ldots$ be an execution of a 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 III.2 (Trace Equivalence and CCDFG Refinement)**.** 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. We say that 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 $\mathcal{E}'$.

Informally, our goal in certifying that a behavioral 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" [13], [14] $\tau \triangleq \langle pre, \mathcal{T}, post \rangle$. Informally, τ is applicable to a CCDFG which satisfies *pre* and produces a CCDFG which satisfies *post*. This notion is formalized below.

Definition III.3 (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 and justifies decomposition of a transformation into a sequence of primitive transformations.

Theorem III.1 (Correctness of Transformation Sequences)**.** Let $\tau_0, \tau_1, \ldots, \tau_n$ be a sequence of correct transformations, where $\tau_i \triangleq \langle pre_i, \mathcal{T}_i, post_i \rangle$. Furthermore, for each $1 \leq i < n$, $post_i \Rightarrow pre_{i+1}$. Then the transformation $\langle pre_1, \mathcal{T}, post_n \rangle$ is correct.

Remark. For the reader familiar with ACL2, it is instructive to understand how Theorem III.1 is formalized. Note that the statement of the theorem involves an arbitrary sequence of *pre* and *post* predicates. A closed-form rendition of the statement involves quantification over functions, which cannot be expressed in the first-order logic of ACL2. Instead, we formalize the statement as a *verification template* as follows. Using encapsulation, we introduce two pairs of guarded transformations $\langle pre_1, \mathcal{T}_1, post_1 \rangle$ and $\langle pre_2, \mathcal{T}_2, post_2 \rangle$ with the associated constraints that each induces a correct transformation, and $post_2 \Rightarrow pre_1$. Then we prove the transitivity theorem $\langle pre_1, \mathcal{T}_1, post_2 \rangle$, which is easy in ACL2. Finally, we develop a macro, which, given a sequence of concrete guarded transformations, repeatedly instantiates the transitivity theorem through functional instantiation.

Our approach to ameliorate the cost of theorem proving is to 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 III.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.

Although the statement of the theorem is simple, the formal proof of its correctness is somewhat nontrivial. The reason is that the formalization requires viewing the transformation as a graph manipulation. As is well-known, reasoning about properties of graph manipulation is complicated [15]. We must also reason about the mapping of operations to graph nodes, and the relation between graph reachability and flow

[2] We say "loosely" since stuttering trace containment is traditionally defined in terms of infinite traces, while our current formalization only allows finite executions 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);
```

Fig. 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.

structure of the underlying program execution. Nevertheless, we believe that the generic nature of the statement ameliorates the verification cost. Furthermore, much of the infrastructure developed in the process is reusable for verification of other transformations. For instance, one side effect is a reusable library of lemmas about graph operations.

Consider the sequence of transformations shown in Fig. 3 for our GCD example. The transformed code conducts two modular divisions in one cycle, thus speed up the computation of GCD. Note that the transformations involved include generic properties of loop unrolling and code motion, together with partial interpretation of two operations; these properties can be easily stated using ACL2. We are currently working on certifying this sequence. The transformations applied in this step may affect the complexity of equivalence checking later (See Section IV).

We end the discussion of the ***verified*** framework with one other 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.

IV. Model Checking

We now discuss the ***verifying*** component of our framework, namely checking equivalence between a CCDFG and the synthesized circuit. We first formulate the notion of equivalence between a CCDFG and a circuit. We then discuss our approach for checking this equivalence.

A. 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 correspondence with CCDFGs.

Definition IV.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}.

Fig. 4 shows a synthesized circuit derived from the CCDFG in Fig. 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.

B. 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.

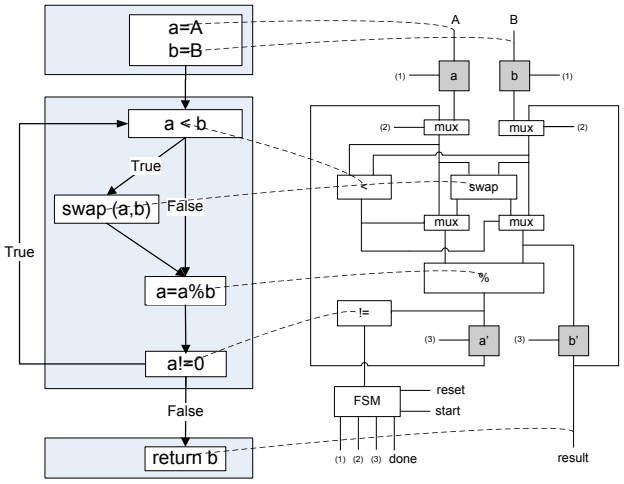

Fig. 4. Synthesized Circuit for GCD and its operation mapping relation with CCDFG in Fig. 2. $A, B, reset, start$ are the input nodes, and the shaded nodes a, a', b, b' are latches. The dotted lines represent the mapping from operations of CCDFG to combinational nodes of circuit. We consider a wire to be a combinational node.

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. 4). 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 IV.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.

C. 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. 5. The upper rail simulates G while the

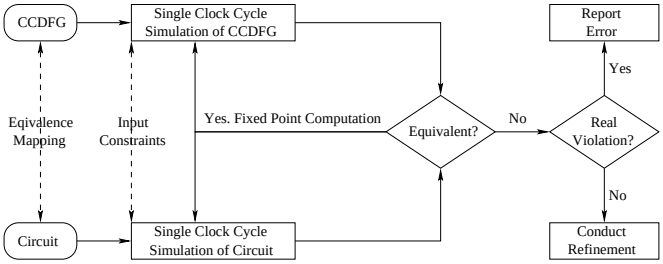

Fig. 5. Dual-Rail Simulation Scheme for Equivalence Checking

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

TABLE I
EQUIVALENCE CHECKING RESULTS FOR GCD UNDER SCHEDULE REFINEMENT

Circuit		Before schedule refn.		After schedule refn.	
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

TABLE II
EQUIVALENCE CHECKING RESULTS FOR GCD UNDER LOOP UNROLLING OPTIMIZATION

	Before trans. seq. in Fig. 3				After trans. seq. in Fig. 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

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 microstep 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. The scheme is analogous to but extends traditional GSTE-style model checking [16] 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.

D. Experimental Results and Discussion

We implemented the above equivalence checking algorithm in the Intel *Forte* environment [17]. The equivalence checking is based on symbolic simulation; symbolic states are represented at bit level using BDDs. We checked the equivalence between a set of CCDFGs of GCD (the original and several transformed versions) and their synthesized circuits. To better comprehend the complexity of equivalence checking, we conduct the experiments without any abstraction. All experiments were conducted on a workstation with 3GHz Intel Xeon processor with 2GB memory.

Table I shows the equivalence checking results for GCD under schedule refinement (Theorem III.2). Since we bit-blast all the operations in the CCDFG, 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. The schedule refinement partitions the loop body into two clock cycles and it does not change the fixed-point computation since the fixed-point is computed based micro-steps. However, since we check the CCDFG-circuit equivalence cycle by cycle, the number of cycles that the circuit is simulated doubles. The running time after schedule refinement is about two times slower than that before for small bit widths. However, for large bit widths, the running time is dominated by the complexity of the CCDFG simulation instead of the circuit simulation. The decreases in running time with the increase in bit width from 7 to 8 are likely due to BDD variable reordering by Forte.

Table II shows the equivalence checking results for GCD under the transformation sequence in Fig. 3. 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 also increases the number of circuit nodes, therefore, increasing the circuit simulation time.

V. RELATED WORK

An early effort [18] on verification of high-level synthesis targets the behavioral portion of VHDL [19]. A formal semantics based on CSP [20] was established for behavioral VHDL. A translation from behavioral VHDL to dependence flow graphs [21] was verified by structural induction based on the CSP semantics. Recently, there has been research on certified synthesis of hardware from formal languages such as HOL [22] 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 specified in Esterel, a synchronous design language, has been also been developed [23] in which a variant of Esterel was embedded in HOL to enable formal reasoning.

Dave [24] provides a comprehensive survey of compiler verification. One of the earliest work on mechanized compiler verification was the Piton project [25], which verified a compiler for an simple assembly language. Compiler certification forms a critical component of the Verisoft project [26], a pervasive formal verification project aimed at ensuring correctness of implementations of critical computing systems with both hardware and software components. The Verifix [27] and

CompCert [28] projects have explored a general framework for certification of compilers targeting various C subsets [29], [30]. There has also been work on a ***verifying*** compiler, where each instance of a compiler transformation generates a proof obligation discharged by a theorem prover [31].

There has been much research on sequential equivalence checking (SEC) between RTL and gate-level hardware designs [32], [33]. Research has also be done on combinational equivalence checking between high-level designs in software-like languages (e.g., SystemC) and RTL-level designs [34]. There has also been effort for SEC between software specifications and hardware implementations [35]: 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, which preserve both control/data flows and the scheduling information. There has also been work on equivalence checking with other graph representations, e.g., Signal Flow Graph (SFG) [36].

VI. Conclusion

We have described 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 efficacy 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. 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. The transformed CCDFG can then be used for equivalence checking with the synthesized design.

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 [4] or xPilot [3]. In the ***verified*** component, we are formalizing other generic transformations including code motion across different loop iterations. In the ***verifying*** component, we are considering three approaches for complexity reduction: (i) use theorem proving to partition a CCDFG into smaller independent CCDFGs and check them individually; (ii) find an efficient symbolic indexing scheme for CCDFG to reduce the number of symbolic variables used in BDD-based equivalence checking; (iii) lift part of the operations into word-level to further reduce the complexity.

References

[1] "Forte Design Systems. Behavioral Design Suite," see URL: `http://www.forteds.com`.

[2] Celoxica, "DK design suite," see URL: http://www.celoxica.com.

[3] 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.

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

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

[6] "Cryptol: The Language of Cryptography," see URL: `http://www.cryptol.net`.

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

[8] M. Kaufmann and J. S. Moore, "Structured Theory Development for a Mechanized Logic," *Journal of Automated Reasoning*, vol. 26, no. 2, pp. 161–203, 2001.

[9] M. Abadi and L. Lamport, "The Existence of Refinement Mappings," *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, May 1991.

[10] L. Lamport, "What Good is Temporal Logic?" *Information Processing*, vol. 83, pp. 657–688, 1983.

[11] P. Manolios, K. Namjoshi, and R. Sumners, "Linking Model-checking and Theorem-proving with Well-founded Bisimulations," in *CAV*, 1999.

[12] R. Sumners, "An Incremental Stuttering Refinement Proof of a Concurrent Program in ACL2," in *ACL2 Workshop*, Austin, TX, Oct. 2000.

[13] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–583, 1969.

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

[15] J. S. Moore, "An Exercise in Graph Theory," in *Computer-Aided Reasoning: ACL2 Case Studies*, M. Kaufmann, P. Manolios, and J. S. Moore, Eds. Boston, MA: Kluwer Academic Publishers, June 2000, pp. 31–58.

[16] J. Yang and C.-J. H. Seger, "Introduction to generalized symbolic trajectory evaluation," *Transaction on VLSI Systems*, vol. 11, no. 3, June 2003.

[17] C.-J. Seger, R. Jones, J. O'Leary, T. 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*, vol. 24, no. 9, 2005.

[18] R. O. Chapman, "Verified high-level synthesis," Ph.D. dissertation, Ithaca, NY, USA, 1994.

[19] IEEE, "IEEE Std 1076: IEEE standards VHDL language reference manual."

[20] *Communicating Sequential Processes*. Prentice-Hall, 1985.

[21] R. Johnson and K. Pingali, "Dependence-based program analysis," in *PLDI*, 1993, pp. 78–89.

[22] M. Gordon, J. Iyoda, S. Owens, and K. Slind, "Automatic formal synthesis of hardware from higher order logic," *Electr. Notes Theor. Comput. Sci.*, vol. 145, pp. 27–43, 2006.

[23] K. Schneider, "A verified hardware synthesis for Esterel," in *Workshop on Distributed and Parallel Embedded Systems (DIPES)*, F. Rammig, Ed. Kluwer, 2000, pp. 205–214.

[24] M. A. Dave, "Compiler verification: a bibliography," *SIGSOFT Software Engineering Notes*, vol. 28, no. 6, p. 2, 2003.

[25] J. S. Moore, *Piton: A Mechanically Verified Assembly Language*. Kluwer Academic Publishers, 1996.

[26] T. V. Consortium, "The verisoft project," http://www.verisoft.de.

[27] "The verifix project," http://www.info.uni-karlsruhe.de/ verifix.

[28] "The compcert project," http://pauillac.inria.fr/ xleroy/compcert.

[29] D. Leinenbach, W. J. Paul, and E. Petrova, "Towards the formal verification of a C0 compiler: Code generation and implementation correctnes," in *SEFM*, 2005, pp. 2–12.

[30] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," in *POPL*, 2006, pp. 42–54.

[31] L. Pike, M. Shields, and J. Matthews, "A Verifying Core for a Cryptographic Language Compiler," in *ACL2 Workshop*, 2006.

[32] 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.

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

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

[35] X. Feng, A. J. Hu, and J. Yang, "Partitioned model checking from software specifications," in *ASP-DAC*, 2005, pp. 583–587.

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