Validating Scheduling Transformation for Behavioral Synthesis

Zhenkun Yang*, Kecheng Hao*, Kai Cong*, Li Lei*, Sandip Ray† and Fei Xie*

* Dept. of Computer Science, Portland State University, Portland, OR 97207, USA

{zhenkun, kecheng, congkai, leil, xie}@cs.pdx.edu

† Strategic CAD Labs, Intel Corporation, Hillsboro, OR 97124, USA

sandip.ray@intel.com

***Abstract*—Behavioral synthesis automatically compiles an electronic system-level description of a hardware design into an RTL implementation. Scheduling in behavioral synthesis is an important, sophisticated, and error-prone transformation which converts the untimed or partially timed description into a fully timed implementation. We present a scalable equivalence checking algorithm for validating scheduling transformations. Our approach accounts for control/data dependency, scheduling modes, and subtle interface protocols. We successfully validated designs with tens of thousands of lines of RTL synthesized by commercial synthesis tool, demonstrating the viability of our approach.**

I. Introduction

Behavioral synthesis is the process of translating a high-level description of a hardware design (usually in C, C++, or SystemC) into a register-transfer level (RTL) implementation. It allows hardware designers to develop systems at a higher abstraction level than directly implementing them in RTL, thereby significantly improves design productivity. However, a critical requirement for practical adoption of behavioral synthesis is verification support to ensure correspondence between the high-level description and the synthesized RTL. However, this is a complex enterprise, since the high abstraction difference between high-level models and RTL makes it difficult to compare them directly through sequential equivalence checking (SEC) techniques [1], [2]. In particular, there is no direct mapping between internal variables of the two, making it difficult to apply traditional SEC decompositions.

Previous work [3]–[5] made partial progress in providing SEC for behavioral synthesis. SEC frameworks were developed for front-end compiler transformations and back-end code-generation modules. Both front-end and back-end SEC were shown to scale to complex industrial-size designs.

However, SEC in previous work did not handle scheduling transformations, *i.e.*, transformations that generate fully timed implementations from untimed or partially timed descriptions. Unfortunately, scheduling is also one of the most complex activities in behavioral synthesis, since it performs aggressive optimizations to meet timing and resource constraints. In fact, the complexity of scheduling transformations distinguishes behavioral synthesis from both software compilers and hardware logic synthesis.

In this paper, we present an approach to validating scheduling transformations in behavioral synthesis. Our approach involves careful characterization of different scheduling modes, formalization of the notions of equivalence to compare designs scheduled according to each mode, and equivalence checking algorithms to perform such comparison. We used our approach on two behavioral synthesis benchmarks, CHStone [6] and S2CBench [7], containing designs from as variety of application domains. The designs were synthesized by a state-of-the-art commercial behavioral synthesis tool. Some of the designs in the benchmark synthesized to tens of thousands of lines of RTL. Our approach successfully validated scheduling transformations in all designs in a few seconds, demonstrating the scalability of our approach. Furthermore, we found several bugs in the synthesis tool itself. The experience underlines both the feasibility and the critical need for such an equivalence checking framework to ensure high-quality synthesized hardware designs.

II. Background

A. Behavioral Synthesis

Figure 1 shows the typical flow of a behavioral synthesis tool. It accepts a behavioral level design written in either untimed C/C++ or timed SystemC, builds an intermediate representation (IR) through its compiler front-end, and applies a sequence of transformations which can be classified into the following categories.

1) *Compiler transformation* includes generic compiler optimizations, *e.g.*, dead code elimination, common subexpression elimination, loop unrolling etc.
2) *Scheduling and binding*. Scheduling assigns every operation a clock cycle to execute, and ensures that the result satisfies timing and resource constraints. In this phase, each operation maps to a hardware resource (*e.g.*, the operation '$\star$' is bound to a multiplier).
3) *Code generation* involves creating RTL implementation code (*e.g.*, VHDL, Verilog, etc.) from the IR.

B. SEC for Back-End and Front-End Transformations

Fig. 1 shows an illustrative SEC framework for behavioral synthesis developed in previous research. Back-end algorithms were defined for comparing the synthesized RTL with the IR generated by synthesis tools after compiler and scheduling transformation [3], [4]; they exploit the operation binding computed by the synthesis tool to define mappings between

978-3-9815370-6-2/DATE16/©2016 EDAA

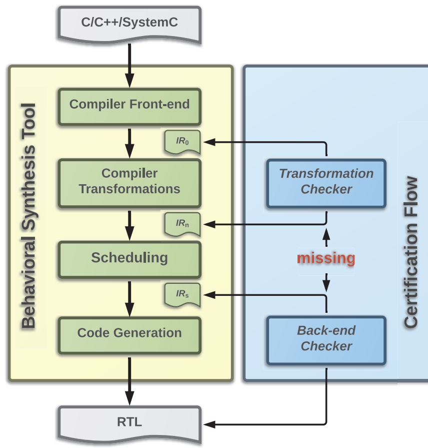


Fig. 1: Behavioral synthesis and certification flow

operations in the IR and RTL. Separately, there was also recent work on SEC for front-end compiler transformations [5]. It uses symbolic simulation to compare IRs generated after each successive transformation.

However, the above frameworks do not support scheduling transformations, resulting in a serious hole in the certification flow for behavioral synthesis. Furthermore, neither the back-end nor the front-end techniques can be easily extended to adapt to scheduling. Back-end techniques critically depend on operation mapping through binding stages of behavioral synthesis which occur later in the synthesis flow than scheduling. On the other hand, optimizations used for front-end transformations (*e.g.*, inductive assertions) cannot be directly applied to provide a guarantee for timing correlation between timed and untimed designs.

III. Scheduling Transformation

Scheduling is a critical synthesis phase that directly affect the quality of synthesized design in terms of timing, performance, and thermal characteristics [8]. Scheduling transformation involves complex heuristics to ensure that the design being synthesized can meet the timing and resource constraints while preserving control and data dependencies.

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

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

Assume that the design is non-pipelined and needs to be implemented with a single 3-cycle multiplier and two adders. Suppose the scheduling transformation schedules **S1** to start at clock cycle t. Since there is data dependency between **S1** and **S2**, and the multiplier takes 3 clock cycles, **S2** cannot be scheduled to start before cycle $t+3$. Furthermore, since there is only one multiplier, **S3** cannot be scheduled to start before cycle $t+3$ (or after $t-3$) either, although there is no data dependency between **S1** and **S3**. However, since there are two adders, **S2** and **S4** can be scheduled concurrently.

For untimed `C/C++` designs, scheduling can assign any clock cycle to an I/O operation as long as control/data dependencies and resource constraints as discussed above are met. However, most high-level descriptions of hardware designs also specify partial timing. For example, SystemC designs can have default I/O timing constraints that are usually specified by `wait` statements. Behavioral synthesis tools usually provide the user the flexibility to explore different architectures by picking different I/O scheduling modes [9] as discussed below.

1) *Cycle-Fixed mode*: In this mode, the user explicitly specifies the timing of the I/O operations, and the scheduling transformation cannot change or refine this timing. This is applied typically at design interfaces which implement communication protocols (possibly with other external interfaces), and the cycles when data must be read or written is governed by the protocol.
2) *Superstate-Fixed mode*: In this mode, the user specifies `wait` statements. The scheduler comprehends these `wait` statements to be the boundaries of "superstates", which impose constraints on scheduling I/O operations as follows. Informally, a superstate is a sequence of operations, possibly scheduled over multiple clock cycles, with the requirements that (1) no I/O operation in a superstate can be moved across the superstate boundary, and (2) all I/O writes must be scheduled at the last clock cycle assigned to operations in the superstate.
3) *Free-Floating mode*: The scheduling transformation can assign any I/O operation to any clock cycle (possibly switching their program order), even add or delete clock cycles, as long as control/data dependencies are maintained.

IV. Problem Formalization

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

Conventions. Our formalization of IRs uses the Control/Data Flow Graph (CDFG). We assume that (1) the set V_o of operations in an IR is a subset of a fixed set O of all operations; and (2) all operations in O are defined through operational semantics over abstract machine states. The IR is decomposed into a collection of basic blocks V_b. These assumptions are standard in formalization of control constructs for programming language semantics. Furthermore, O is assumed to contain standard variable *read* and *write* operations with the usual meaning, and a *wait* operation that specifies a transition with no effect on machine state. Given a set V of operations, we define the *active subset* of V, denoted by $V[N]$ to be the subset of V excluding all wait statements. For each operation $o \in V_o$ we assume there is a unique basic block in V_b

containing o; this is easily implemented by uniquely labeling all design operations. Following conventions from program analysis, the control flow graph of an IR is a directed graph $G_C \triangleq (V_b, E_c)$ where an edge $e \in E_c$ from basic block b_0 to b_1 represents a control dependency of b_1 on b_0, and the data flow graph is the directed graph $G_D \triangleq (V_o, E_d)$ where an edge $e \in E_d$ from operation o_1 to o_2 represents a data dependency of o_2 on o_1.

Definition 1 (CDFG)
The CDFG is a triple $G \triangleq (G_C, G_D, R)$, where $G_c \triangleq (V_b, E_c)$ is a control flow graph, $G_D \triangleq (V_o, E_d)$ is a data flow graph, and R is a mapping $R : V_o \rightarrow V_b$.

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

We formalize the timing associated with an operation execution with the notion of a *state transition partition* (STP) below. It is convenient to interpret the pair (P_i, τ_i) as the directive that (active) operations in P_i are scheduled at cycle τ_i.

Definition 2 (STP)
Let V_o be a set of operations. A state transition partition *of V_o is a finite set of pairs $\{(P_i, \tau_i) : i = 1 \dots k\}$, where each P_i is a sequence of operations over $V_o[N]$ and the following conditions hold:*

1) $\bigcup_{i=1}^{k} P_i = V_o[N]$;
2) $P_i \cap P_j = \emptyset$ *for* $i \neq j$;
3) $\tau_i \in \mathbb{N}$ *with* $\tau_i \neq \tau_j$ *for* $i \neq j$.

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

It is convenient to view each partition P_i as a sequence requiring that if an operation o appears before o' then o' cannot be scheduled for execution before o. We utilize this restriction in defining trace compatibility below. Informally, an operation o can be scheduled at cycle τ only after any operations o depends on have completed, either in a previous cycle or earlier in the same cycle.

Definition 3 (Operation Precedence)
Given an STP S over a set of operations V_o, and two operations $o_1, o_2 \in V_o$, we say o_2 follows *o_1 in V_o if either (1) $\tau[o_2] > \tau[o_1]$, or (2) $\tau[o_1] = \tau[o_2]$ and o_2 appears after o_1 in $P[o_1]$.*

Definition 4 (Trace Compatibility)
Let $G \triangleq (G_C, G_D, R)$ be a CDFG over the set of operations V_o and basic blocks V_b, and let $S \triangleq \{(P, \tau)\}$ be an STP over V_o. We say that S is compatible *with G if the following conditions hold for each pair of operations o_i and o_j in $V_o[N]$:*

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

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

Definition 5 (Valid Cycle-Fixed Schedule)
Let $S \triangleq \{(P_i, \tau_i), i = 1, \dots, k\}$ over an operation set V_o and $G \triangleq (G_C, G_D, R)$ be a CDFG. We say that S is a valid cycle-fixed schedule *with respect to G if S is compatible with G, and the following additional condition holds:*

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

Definition 6 (Valid Superstate-Fixed Schedule)
Let $S \triangleq \{(P_i, \tau_i), i = 1, \dots, k\}$ over an operation set V_o and $G \triangleq (G_C, G_D, R)$ be a CDFG. We say that S is a valid superstate-fixed schedule *with respect to G if S is compatible with G, and additional timing conditions hold which are specified as follows. Let o_i and o_j be two read or write operations such that $R(o_i) = R(o_j)$. Suppose that there is a path Π in G_D from o_i to o_j that has n `wait` operations. Then $\tau[o_j] \geq \tau[o_i] + n$. In addition, let o_1, o_2, o_3 be operations such that $R(o_1) = R(o_2) = R(o_3)$, o_1 and o_2 are write operations, and o_3 is a `wait` operation. Suppose that there are paths Π_1 and Π_2 in G_D from o_1 to o_3 and o_2 to o_3 such that there is no intermediate wait operation. Then:*

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

Requirements 1-3 above ensure that the scheduling does not "squeeze" I/O operations by removing clock cycles. In addition, any write operation is scheduled in the last cycle before any user-provided wait operation.

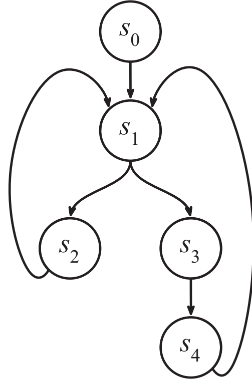
V. Validation Approach

A. Validating Trace Compatibility

Let G be the CDFG of a design and S be the STP after scheduling for a set V_o of operations. Our approach to control/data dependency checking (and hence free-floating mode scheduling) is to first define a *dependency graph* G_Δ, consolidating the control and data dependencies. The dependency graph $G_\Delta = (V_\Delta, E_\Delta)$ is a pair, where V_Δ is a set of operations, E_Δ is a list of tuples. Each tuple $\langle o_i, o_j, \mathcal{C} \rangle$ means that operation o_j depends on o_i under condition $\mathcal{C}$, where $\mathcal{C}$ is a conjunction of Boolean expressions. G_Δ not only captures the data dependencies of a design, but also includes control dependencies through the condition encoded in each edge. Constructing G_Δ requires a traversal of G, and identifications of each operation $o \in V_o$ and the condition under which

```
void dut::my_thread() {
  initialize();
  while( true ) {
    wait();
    a = 1;
    if(x) {
      wait();
      b = a + 1;
    } else {
      wait();
      c = a + b;
      wait();
      d = c + 1;
    }
  }
}
```

(a)



(b)

Fig. 2: Extract *superstates* of a thread in SystemC. (a) `my_thread` is a thread of module `dut`, `wait` statements are the boundary of superstates in SystemC. (b) Superstates and their transitions of `my_thread`.

o is executed. This is done efficiently using *def-use* chain analysis [10]: since the left-hand-side of every assignment is unique we can trivially identify variable dependency chains. Finally, to check if the control/data dependencies are satisfied in S, it is sufficient that for each pair of operations o_i and o_j, o_j follows o_i in S under condition $\mathcal{C}$.

From Definition 4, S is *compatible* with G if control and data dependencies are preserved in S. We can validate this condition by comparing the dependency graphs of S and G.

B. Validating I/O Timing

Algorithm 1: CHECK-CYCLE-FIXED-MODE(G, S)

1 $ss \leftarrow$ BUILD-SUPERSTATES(G)
2 $\mathcal{T}_1 \leftarrow$ COMPUTE-TRACES(ss)
3 $\mathcal{T}_2 \leftarrow$ COMPUTE-TRACES(S)
4 **foreach** $\pi = [s_1^s, s_2^s, \ldots, s_m^s] \in \mathcal{T}_1$ **do**
5 $\quad C \leftarrow$ EXTRACT-TRACE-COND(π)
6 $\quad [s_1^g, s_2^g, \ldots, s_n^g] \leftarrow$ FIND-TRACE($\mathcal{T}_2, C$)
7 $\quad$ **assert** $m = n$ $\quad \triangleright$ *have the same number of cycles*
8 $\quad$ **for** $i \leftarrow 0$ **to** m **do**
9 $\quad\quad$ **assert** HAS-1-1-MAPPING($\mathcal{F}_{rw}(s_i^s),\ \mathcal{F}_{rw}(s_i^g)$)
10 **return true**

For Cycle-Fixed and Superstate-Fixed modes, we must additionally validate the I/O timings. It will be convenient to call the partitions in the STP S to be *states*, and the set of operations between any two user-specified `wait` operations in a CDFG G to be *superstates*. Fig. 2(a) shows an example of a module `dut` in SystemC, where `my_thread` is a thread of `dut`. Fig. 2(b) shows the superstates and their transitions in `my_thread`. Let an execution trace $\pi = [s_1^s, s_2^s, \ldots, s_i^s, \ldots]$ of a CDFG $G \triangleq (G_C, G_D, R)$ be a sequence of execution of superstates following control flow in G_C. Let an execution trace $\pi' = [s_1^g, s_2^g, \ldots, s_j^g, \ldots]$ of an STP S be a sequence of operation segments, such that operations in each segment belong to the same partition. Since we have already checked the control and data dependencies of G and S in Section V-A, we know loop structures in π and π' are preserved by the scheduling transformation. Thus, when computing the traces of G and S we can break loop back-edges temporarily. As a result, traces in G and S are finite.

From the requirements of Cycle-Fixed mode, the number of superstates after scheduling must be equal to the number of scheduled states. Algorithm 1 checks if STP S is a valid Cycle-Fixed scheduling of CDFG G. Function BUILD-SUPERSTATES build superstates ss from G. The superstates can be obtained easily by traversing G in a *depth-first search* manner while accumulating operations. A new superstate is built when we encounter a `wait` operation. Then function COMPUTE-TRACES computes all traces in ss and S. Since we temporarily break loop back-edges, there is a finite number of traces for ss and S. For each trace $\pi \in \mathcal{T}_1$, functions EXTRACT-TRACE-COND and FIND-TRACE pair up traces in ss and S according to the trace condition C. We `assert` that each paired traces should be executed in the same number of cycles, and within each cycle, function HAS-1-1-MAPPING checks that I/O operations have one-to-one mappings between each superstate s_i^s and operation segment s_i^g.

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

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

Algorithm 2 checks if STP S is a valid Superstate-Fixed scheduling of CDFG G. Each superstate may be "stretched" into multiple states after scheduling. For each trace pair $\pi \in \mathcal{T}_1$ and $\pi' \in \mathcal{T}_2$, we use function FIND-SHORTEST-SEGMENTS to find the corresponding trace segments that were "stretched" from a particular superstate. Function FIND-SHORTEST-SEGMENTS($\pi', start, s_i^s$) finds the trace segments $seg = s_{start}^g, \ldots, s_{end}^g$ with minimum length which starts from $start$ and ends at end, such that $\mathcal{F}_{io}(s_i^s) \subseteq \mathcal{F}_{io}(seg)$. We then `assert` that I/O operations have one-to-one mappings between each s_i^s and trace segments seg, which means that

all I/O operations are within the bound of the superstate.

VI. Experimental Results

TABLE I: Summary of Evaluation on S2CBench Benchmark

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

TABLE II: Summary of Evaluation on CHStone Benchmark

App. Domain	Design	Lines of code C	Lines of code RTL	Time (s)
Arithmetic	DFADD	542	12933	0.11
	DFDIV	452	10948	0.13
	DFMUL	392	7100	0.07
	DFSIN	772	22949	0.34
Microprocessor	MIPS	256	7237	0.05
Media Processing	ADPCM	521	33706	0.84
	GSM	388	22816	0.45
	JPEG	1031	53584	1.46
	MOTION	414	13770	1.38
Security	AES	699	40014	1.37
	BLOWFISH	1241	23490	0.53
	SHA	1284	12491	0.13

We applied our algorithms on two behavioral synthesis benchmarks, CHStone [6] and S2CBench [7]. The designs were synthesized by a commercial synthesis tool, and many of them generated tens of thousands of lines of RTL. The necessary CDFGs and STPs are created by parsing the reports generated by the synthesis tool. Since designs in the CHStone benchmark do not have *wait* statements, the only scheduling mode is free-floating. However, the benchmark is still illustrative since the designs are large and complex. S2CBench has SystemC designs with *wait* statements, permitting us to exercise algorithms for different scheduling modes as well. We conducted our experiments on a workstation with Debian 7.1 running on a 2.93 GHz Intel Xeon X3470 processor with 8GB memory. Tables I and II summarize the results. Note that we can validate all the designs from both benchmarks within 10 seconds. For S2CBench benchmark, the `FFT` design is not shown, because the floating point datatype in it is not accepted by the synthesis tool.

Interestingly, we found two bugs in the synthesis tool itself, based on violation of SystemC specifications. For pedagogical reasons, we provide simplified version of the programs and show both pre- and post-scheduling designs in SystemC.

Fig. 3(a) shows a design before scheduling, where `Out` is an output signal of type `sc_uint<16>`. According to the specification, the write statement `Out.write(a)` in line 5 is invisible and invalid, therefore should be eliminated; the only observable behavior should be the write statement in line 6. However, as shown in Fig. 3(b), the scheduling transformation scheduled the two writes into two different states. Thus, both writes are observable, which violates the SystemC standard. Algorithm 2 detects this violation by checking that the two `write` statements are scheduled to two different states.

Fig. 4 shows another scheduling bug. In Fig. 4(a), signal `sig` is written and read in the same cycle. Therefore variable `j` will take the old value of `sig`. However, after scheduling, as shown in Fig. 4(b), the read of `sig` is scheduled to the next cycle after the write. Variable `j` will take the new value of `sig` instead of the old one.

VII. Related Work

Koelbl *et al.* [11] provide a comprehensive tutorial on methods of equivalence checking of high-level designs with RTL. Anderson [12] reports an early effort on the verification of *as soon as possible* scheduling transformation using theorem proving. Narasimhan *et al.* [13] used theorem proving approach to verification of force-directed list scheduling algorithm for resource-constrained scheduling in high-level synthesis. Karfa *et al.* [14] develops techniques for more automated SEC on scheduling transformation. This framework converts the designs before and after scheduling transformation into Finite State Machine with Datapath (FSMD) models, then checks the equivalence of two FSMD models. The major difference between the above approaches and our research is the observation that scheduling transformations can be extricated from compiler transformations and handled as a verification of partitioning. This permits efficient static checking to validate these transformations, obviating expensive theorem proving or symbolic simulation techniques used in previous work. The efficiency is critical in enabling application of our approach on large-scale designs. Finally, the the SLEC equivalence checker from Calypto is reported to be able to handle equivalence checking under timing constraints [15]; however, no details are provided on the approach, and it is unclear whether different scheduling modes are handled.

VIII. Conclusions and Future Work

We have developed an efficient approach to validating scheduling transformations in behavioral synthesis. We characterized different scheduling modes, formalized equivalence relations to compare designs scheduled by different modes, and proposed efficient algorithms to validate designs scheduled by each mode. Experiments on synthesizable designs in S2CBench show that our approach can successfully validate all designs within a few seconds. Furthermore, our approach detected bugs in a commercial behavioral synthesis tool.

For future work, in addition to checking I/O timing constraints, we will check resource constraints in characterization

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

(a)

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

(b)

Fig. 3: An example of incorrect scheduling of signal I/O. (a) Design before scheduling, where a signal output `Out` is written twice with different values, however, only the last write is valid. (b) Design after scheduling, where two writes of `Out` are scheduled to two different cycles. We use `add_state()` to denote that scheduling transformation will add a new state.

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

(a)

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

(b)

Fig. 4: An example of incorrect scheduling of signal I/O. (a) Design before scheduling, where `sig` is written and then read at the same cycle. The read statement takes the old value. (b) Design after scheduling, where the write and read statements are scheduled to two different cycles. We use `add_state()` to denote that the scheduling transformation will add a new state.

results of the technology libraries used by scheduling transformation. Furthermore, behavioral synthesis tools often map high-level data structures on the interface to different predefined interface components.*e.g.*, an interface variable can be synthesized to components with handshaking protocols, memories, FIFOs, and Modular interfaces. It will be interesting to validate the correctness of interface synthesis process.

References

[1] P. Chauhan, D. Goyal, G. Hasteer, A. Mathur, and N. Sharma, "Non-cycle-accurate sequential equivalence checking," in *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, July 2009, pp. 460–465.

[2] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley, "Solver technology for system-level to rtl equivalence checking," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, April 2009, pp. 196–201.

[3] S. Ray, K. Hao, Y. Chen, F. Xie, and J. Yang, "Formal verification for high-assurance behavioral synthesis," in *Proc. of ATVA*, 2009, pp. 337–351.

[4] K. Hao, F. Xie, S. Ray, and J. Yang, "Optimizing equivalence checking for behavioral synthesis," in *Proc. of DATE*, 2010, pp. 1500–1505.

[5] Z. Yang, K. Hao, K. Cong, L. Lei, S. Ray, and F. Xie, "Scalable certification framework for behavioral synthesis front-end," in *Proc. DAC*, 2014, pp. 149:1–149:6.

[6] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical c-based high-level synthesis," *Information and Media Technologies*, vol. 4, no. 4, pp. 740–752, 2009.

[7] B. Schafer and A. Mahapatra, "S2CBench: Synthesizable SystemC benchmark suite for high-level synthesis," *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 53–56, Sep. 2014.

[8] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *Proceedings of the 43rd annual Design Automation Conference*, ser. DAC '06. New York, NY, USA: ACM, 2006, pp. 433–438. [Online]. Available: http://doi.acm.org/10.1145/1146909.1147025

[9] J. P. Elliott, *Understanding Behavioral Synthesis: A Practical Guide to High-Level Design*, 1999th ed. Boston: Kluwer Academic Publishers, May 1999.

[10] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 1st ed. Addison Wesley, Jan. 1986.

[11] A. Koelbl, Y. Lu, and A. Mathur, "Embedded tutorial: formal equivalence checking between system-level models and RTL," in *ICCAD*, 2005, pp. 965–971.

[12] D. Anderson and J. Ainscough, "The verification of scheduling algorithms," in *IEE Colloquium on Structured Methods for Hardware Systems Design*, 1994, pp. 7/1–7/5.

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

[14] C. Karfa, D. Sarkar, C. Mandal, and P. Kumar, "An Equivalence-Checking Method for Scheduling Verification in High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 3, pp. 556–569, Mar. 2008.

[15] Calypto, *SLEC HLS, http://calypto.com/en/products/slec/slec_system-hls*, (accessed April 18, 2015).