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 sub-expression 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 ‘*’ 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
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 affects 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**: \( x = y \times z \)
- **S2**: \( p = x + y \)
- **S3**: \( w = a \times b \)
- **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 \( V_0 \) 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 \( G_C \triangleq (V_c, E_c) \) 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 \( G_D \triangleq (V_d, E_d) \) 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 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 \( o_1 \) and \( o_2 \) in \( V_o[N] \):

1. If there is a path from \( o_1 \) to \( o_2 \) in \( G_D \) then \( o_2 \) follows \( o_1 \) in \( \tau \).
2. If there is a path from \( R(o_1) \) to \( R(o_2) \) in \( G_C \) then \( o_2 \) appears after \( o_1 \) in \( \tau \).

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 \( S \triangleq \{(P_i, \tau_i), i = 1, \ldots, k\} \) 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 \( \Pi \) 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 \( S \triangleq \{(P_i, \tau_i), i = 1, \ldots, k\} \) 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 constraints hold which are specified as follows. 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 \( \Pi \) in \( G_D \) from \( o_1 \) to \( o_2 \) that has \( n \) wait operations. Then \( \tau[o_2] \geq \tau[o_1] + n \).

In addition, let \( o_1, o_2, o_3 \) be operations such that \( R(o_1) = R(o_2) = R(o_3) \), and \( o_1 \) and \( o_2 \) are write operations, and \( o_3 \) is a wait operation. Suppose that there are paths \( \Pi_1 \) and \( \Pi_2 \) in \( G_D \) from \( o_1 \) 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 \( \Pi \) 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 \( \Pi \) from \( o_1 \) (resp., \( o_2 \)) to \( o \) in \( G_D \), or (2) there is a path \( \Pi' \) from \( R(o_1) \) to \( R(o) \) in \( G_C \) (resp., \( o_2 \)). Then \( \tau[o_2] \leq \tau[o_1] \) 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 \( G_\Delta \), consolidating the control and data dependencies. The dependency graph \( G_\Delta = (V_\Delta, E_\Delta) \) is a pair, where \( V_\Delta \) is a set of operations, \( E_\Delta \) is a list of tuples. Each tuple \( (o_i, o_j, \mathcal{C}) \) means that operation \( o_j \) depends on \( o_i \) under condition \( \mathcal{C} \), where \( \mathcal{C} \) is a conjunction of Boolean expressions. \( G_\Delta \) not only captures the data dependencies of a design, but also includes control dependencies through the condition encoded in each edge. Constructing \( G_\Delta \) requires a traversal of \( G \), and identifications of each operation \( o \in V_o \) and the condition under which
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 $C$.

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. $ss \leftarrow \text{Build-Superstates}(G)$
2. $T_1 \leftarrow \text{Compute-Traces}(ss)$
3. $T_2 \leftarrow \text{Compute-Traces}(S)$
4. foreach $\pi = [s_1^g, s_2^g, \ldots, s_m^g] \in T_1$ do
   5. $C \leftarrow \text{Extract-Trace-Cond}(\pi)$
   6. $[s_1^w, s_2^w, \ldots, s_n^w] \leftarrow \text{Find-Trace}(T_2, C)$
   7. assert $m = n$ \ ensure the same number of cycles
   8. for $i \leftarrow 0$ to $m$ do
      9. assert HAS-1-1-MAPPING$(F_{rw}(s_i^w), F_{rw}(s_i^g))$
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 $\pi = [s_1^t, s_2^t, \ldots, s_t^t, \ldots]$ 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 $\pi' = [s_1^g, s_2^g, \ldots, s_m^g, \ldots]$ 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 $\pi$ and $\pi'$ 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 COMPUTETRACES 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 $\pi \in T_1$, 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 $s_i^t$ and operation segment $s_i^g$.

**Algorithm 2: Check-Superstate-Fixed-Mode(G, S)**

1. $ss \leftarrow \text{Build-Superstates}(G)$
2. $T_1 \leftarrow \text{Compute-Traces}(ss)$
3. $T_2 \leftarrow \text{Compute-Traces}(S)$
4. foreach $\pi = [s_1^g, s_2^g, \ldots, s_m^g] \in T_1$ do
   5. $C \leftarrow \text{Extract-Trace-Cond}(\pi)$
   6. $\pi' = [s_1^w, s_2^w, \ldots, s_n^w] \leftarrow \text{Find-Trace}(T_2, C)$
   7. start $\leftarrow 0$
   8. for $i \leftarrow 0$ to $m$ do
      9. end $\leftarrow \text{Find-Shortest-Segments}(\pi', \text{start}, s_i^g)$
      10. seg $\leftarrow [s_i^{\text{start}}, \ldots, s_i^{\text{end}}]$
      11. assert HAS-1-1-MAPPING$(F_{rw}(s_i^w), F_{rw}(seg))$
      12. assert HAS-1-1-MAPPING$(F_{w}(s_i^t), F_{w}(s_i^{\text{end}}))$
      13. start $\leftarrow \text{end} + 1$
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 $\pi \in T_1$ and $\pi' \in T_2$, we use function FIND-SHORTEST-SEGMENTS to find the corresponding trace segments that were “stretched” from a particular superstate. Function FIND-SHORTEST-SEGMENTS$(\pi', \text{start}, s_i^g)$ finds the trace segments $seg = [s_i^{\text{start}}, \ldots, s_i^{\text{end}}]$ with minimum length which starts from start and ends at end, such that $F_{iw}(s_i^g) \subseteq F_{iw}(seg)$. We then assert that I/O operations have one-to-one mappings between each $s_i^t$ and trace segments $seg$, which means that
all I/O operations are within the bound of the superstate.

VI. EXPERIMENTAL RESULTS

<table>
<thead>
<tr>
<th>App. Domain</th>
<th>Design</th>
<th>Lines of code</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security</td>
<td>AES CIPHER</td>
<td>429</td>
<td>3941</td>
</tr>
<tr>
<td></td>
<td>KASUMI</td>
<td>415</td>
<td>3602</td>
</tr>
<tr>
<td></td>
<td>MD5C</td>
<td>467</td>
<td>4105</td>
</tr>
<tr>
<td></td>
<td>SONN 3G</td>
<td>522</td>
<td>3121</td>
</tr>
<tr>
<td>Media Processing</td>
<td>QSORT</td>
<td>204</td>
<td>865</td>
</tr>
<tr>
<td></td>
<td>JPEG</td>
<td>269</td>
<td>1191</td>
</tr>
<tr>
<td></td>
<td>ADPCM</td>
<td>270</td>
<td>370</td>
</tr>
<tr>
<td></td>
<td>FIR</td>
<td>176</td>
<td>561</td>
</tr>
<tr>
<td></td>
<td>DECIMATION</td>
<td>422</td>
<td>3267</td>
</tr>
<tr>
<td></td>
<td>INTERPOLATION</td>
<td>231</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>IDCT</td>
<td>450</td>
<td>4266</td>
</tr>
<tr>
<td></td>
<td>DISPARITY</td>
<td>634</td>
<td>4355</td>
</tr>
</tbody>
</table>

TABLE II: Summary of Evaluation on CHStone Benchmark

<table>
<thead>
<tr>
<th>App. Domain</th>
<th>Design</th>
<th>Lines of code</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>DFADD</td>
<td>542</td>
<td>12933</td>
</tr>
<tr>
<td></td>
<td>DFDIV</td>
<td>452</td>
<td>10948</td>
</tr>
<tr>
<td></td>
<td>DF_MUL</td>
<td>392</td>
<td>7100</td>
</tr>
<tr>
<td></td>
<td>DFSIN</td>
<td>772</td>
<td>22949</td>
</tr>
<tr>
<td>Microprocessor</td>
<td>MIPS</td>
<td>256</td>
<td>7237</td>
</tr>
<tr>
<td>Media Processing</td>
<td>ADPCM</td>
<td>521</td>
<td>33706</td>
</tr>
<tr>
<td></td>
<td>GSM</td>
<td>388</td>
<td>22816</td>
</tr>
<tr>
<td></td>
<td>JPEG</td>
<td>1031</td>
<td>53584</td>
</tr>
<tr>
<td></td>
<td>MOTION</td>
<td>414</td>
<td>13770</td>
</tr>
<tr>
<td>Security</td>
<td>AES</td>
<td>699</td>
<td>40014</td>
</tr>
<tr>
<td></td>
<td>BLOWFISH</td>
<td>1241</td>
<td>23490</td>
</tr>
<tr>
<td></td>
<td>SHA</td>
<td>1284</td>
<td>12491</td>
</tr>
</tbody>
</table>

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 CDFIGs 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.
add_state();
added
an
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.

```c
void block::thread1() {
    int accu = 0;
    while(1) {
        Out.write(accu);
        wait();
    }
}
```

(a)

```c
void block::thread2() {
    var = 0; // var is a variable
    while(1) {
        var ++;
        sig.write(var);
        sc_uint<16> j = sig.read();
        Out.write(j);
        wait();
    }
}
```

(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.

```c
void block::thread() {
    int accu = 0;
    while(1) {
        Out.write(accu);
        add_state(); // an extra cycle is added
        Out.write(accu);
        wait();
    }
}
```

(a)

```c
void block::thread2() {
    var = 0; // var is a variable
    sig.write(0); // sig is a signal
    wait();
    while(1) {
        var ++;
        sig.write(var);
        add_state(); // an extra cycle is added
        sig.write(var);
    }
}
```

(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**


