Handling Design and Implementation Optimizations in Equivalence Checking for Behavioral Synthesis

Zhenkun Yang
Portland State University
zhenkun@cs.pdx.edu

Kecheng Hao
Portland State University
kecheng@cs.pdx.edu

Sandip Ray
University of Texas at Austin
sandip@cs.utexas.edu

Fei Xie
Portland State University
xie@cs.pdx.edu

ABSTRACT

Behavioral synthesis involves generating hardware design via compilation of its Electronic System Level (ESL) description to an RTL implementation. Equivalence checking is critical to ensure that the synthesized RTL conforms to its ESL specification. Such equivalence checking must effectively handle design and implementation optimizations. We identify two key optimizations that complicate equivalence checking for behavioral synthesis: (1) operation gating, and (2) global variables. We develop a sequential equivalence checking (SEC) framework to compare ESL designs with RTL in the presence of these optimizations. Our approach can handle designs with more than 32K LoC RTL synthesized from practical ESL designs. Furthermore, our evaluation found a bug in a commercial tool, underlining both the importance of SEC and the effectiveness of our approach.

Categories and Subject Descriptors

B.6.3 [**Design Aids**]: Design Aids—*Automatic synthesis, Optimization, Verification*

General Terms

Algorithms, Performance, Verification

Keywords

Equivalence checking, behavioral synthesis, optimization

1. INTRODUCTION

Electronic System Level (ESL) specifications provide a promising approach to deal with the high complexity of modern VLSI systems: design functionality is specified at a high level of abstraction (*e.g.*, with SystemC, C/C++, or domain-specific language), and compiled by a behavioral synthesis tool to RTL. Several behavioral synthesis tools are commercially available [8, 3, 2, 4]. However, their adoption critically depends on our ability to *certify* the result of synthesis, *i.e.*, ensure that the synthesized RTL conforms to the ESL specification. This task is challenging because of the large difference in abstraction between the two.

In previous work [19, 11], we have developed a sequential equivalence checking (SEC) framework for behavioral synthesis. The key ingredients of the framework were (1) the use of a formal structure, *Clocked Control/Data Flow Graph* (CCDFG) as a uniform design abstraction, (2) a certified sequence of high-level transformations to reduce the abstraction gap, (3) an SEC algorithm based on dual-rail symbolic simulation between CCDFG and RTL, and (4) optimizations that enable compositional application of SEC exploiting internal cutpoints and modular structures. Experimental results reported successful certification of synthesized designs with tens of thousands of lines of RTL for ESL specifications of a number of cryptographic algorithms.

Unfortunately, the above approach cannot directly handle certification of designs from other domains that involve considerably less structure. In particular, one key requirement to achieve compositionality in SEC is the availability of equivalent internal operations or modules between the abstract CCDFG and the corresponding RTL, which are then used as *cutpoints*. However, we found that for many synthesized ESL designs, there are very few internal operations that preserve such equivalence in the presence of design and implementation optimizations, thus undermining compositionality and hence scalability.

In this paper, we present techniques for SEC between ESL designs and synthesized RTL, in the presence of optimizations that violate local equivalences of internal signals. Our framework can handle large synthesized designs from diverse domains, *e.g.*, we could successfully certify all designs in the CHStone benchmark [12] synthesized by a commercial synthesis tool. CHStone is a publicly available C-based ESL benchmark suite containing designs from four different categories; some designs have over 1600 lines of C, and generate over 32K lines of RTL when synthesized. We are not aware of any other SEC framework for behavioral synthesis that can handle synthesized designs of such diversity and scale. As a point of comparison, the framework described in previous work [11] (including all the SEC optimizations but without the techniques presented here) can certify only one of the twelve designs in the benchmark suite.

Our key observation is that there are two key sources of local inequivalence between CCDFG and RTL:

Operation Gating: Behavioral synthesis tools often opti-

DAC 2013 May 29 - June 07, 2013, Austin, Texas, USA

mize the RTL by introducing control structures or "guards" to ensure that certain operations are executed only when their results are relevant to downstream computation, and turned off otherwise. Such gated operations are functionally equivalent to the behavioral specification only under these guards. This makes such an operation difficult to identify; more problematically, it precludes the naive approach of using it as a cutpoint by verifying it in isolation and replacing it with an uninterpreted function in the CCDFG and RTL.

Global Variables: Global variables are used commonly in ESL as a *design optimization*: the user can then define some design functionalities as implicit side effects of other design modules, reducing the lines-of-code in ESL description and thus improving compactness. Unfortunately, global variables break the compositional approach of verifying modules compositionally, since the side effects on these variables must be accounted for during SEC.

Our key contributions are algorithms that enable compositional SEC for behavioral synthesis in the presence of the above optimizations.

1. We develop an algorithm for *relaxed* SEC that includes identification and compositional use of gated variables. The approach tolerates local, "irrelevant" inequivalences between gated variables and their RTL counterparts, as long as the inequivalences are resolved during symbolic simulation of downstream computation.

2. We develop an approach to modeling the side effects of global variables explicitly and show how the approach can then be used with modular analysis.

The algorithms, albeit not individually complex, have been carefully developed to (1) exploit the constraints and invariants available from the behavioral synthesis process, and (2) reinforce the available SEC optimizations, facilitating smooth integration. Finally, we found a subtle bug in an optimization of the behavioral synthesis tool itself, demonstrating both the need for certification of behaviorally synthesized designs and the importance of SEC in general and our framework in particular to achieve such certification.

2. BACKGROUND

2.1 Behavioral Synthesis

A behavioral synthesis tool takes a high-level behavioral description of a design and a library of hardware resources, and generates an RTL implementation. Similar to a generic compiler, it first performs lexical, syntax and semantic analysis, and builds an intermediate representation (IR) of the high-level description. A series of transformations is then applied to the IR, which can be categorized in three phases:

Compiler transformations form the first level. This includes transformations such as dead code elimination, constant propagation, loop unrolling, etc.

Scheduling transformations entail computing for each operation the clock cycle for its execution. The clock cycle must account for constraints in hardware resources as well as control and data flow. These transformations include pipelining loop iterations, grouping independent operations for concurrent execution, etc.

Resource binding and control synthesis maps a hardware resource to each operation, allocates registers for variables used across clock cycles, and generates a finite state machine (FSM) to implement the schedule.

After these transformations, the design can be expressed in RTL. Often manual tweaks are added to optimize for different parameters (*e.g.*, performance, power, etc.).

2.2 A Certification Framework

In previous work [19, 11], we proposed an SEC framework for certifying behaviorally synthesized RTL. A key idea was to apply SEC to compare the RTL with the design representation after high-level transformations have been applied to the ESL description. The framework introduced a formalization called CCDFG as a design abstraction. The CCDFG semantics entail (1) state-based semantics for individual operations, and (2) interpretation of control and data flows and scheduling constructs. The operations supported are those in the LLVM assembly language [17]. High-level transformations are certified by theorem proving. The transformed CCDFG is compared with RTL through SEC via dual-rail symbolic simulation. Three key optimizations were used to improve SEC performance by exploiting compositionality. *Cutpoints* reduce lengths of symbolic expressions by replacing verified sub-circuits with new symbolic values. *Cut-loop* partitions SEC for a loop into three checks to avoid expensive fix-point computation. *Modular analysis* optimizes SEC by replacing verified sub-modules by uninterpreted functions.

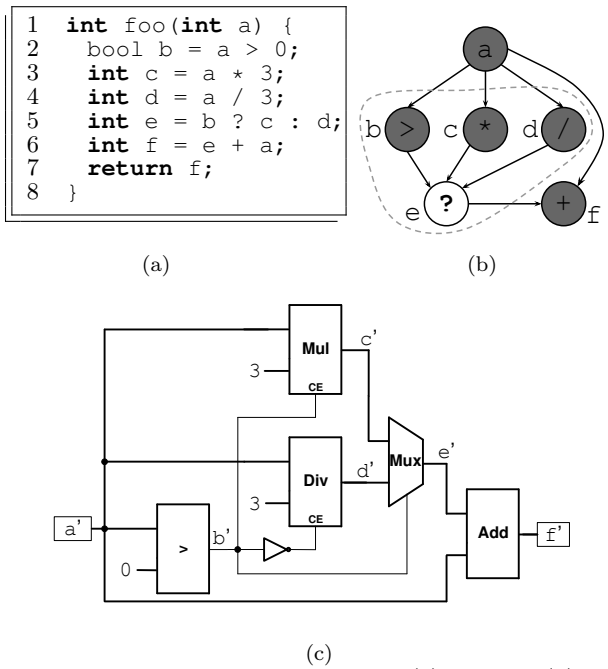
Unfortunately, as we described above, the compositional approaches require equivalence of internal operations or modules between CCDFG and RTL, which is broken in practice by design and implementation optimizations such as global variables and operation gating. Indeed, our attempts to apply the framework on diverse examples showed this was a blocking problem; the SEC optimizations were unusable, and hence the framework did not scale. This motivates developing an SEC approach to robustly handle design and implementation optimizations.

3. CHALLENGES FROM OPTIMIZATION

3.1 Operation Gating

The idea of operation gating is to add controlling predicates so that an operation is not executed when the value computed is irrelevant to downstream computation. Behavioral synthesis tools generate optimized RTL with operation gating to facilitate power-friendly hardware systems [7]. The transformation itself is complex, and its details are not germane to this paper. The characteristic of operation gating that is relevant to equivalence checking is that some operations have explicitly generated gating predicates in the synthesized RTL, when no such predicate appears in the CCDFG. The effects of the operation on the CCDFG and the RTL are then equivalent only when the gating predicate holds.

Consider synthesizing the code fragment shown in Fig. 1(a). According to the semantics of C, the multiplication operation in Line 3 (and the assignment of the result to c) must be executed regardless of the value of b. However, the result of multiplication is only relevant to the eventual return value f when the value of b is 1. In the RTL shown in Fig. 1(c),



(c)

Figure 1: Operation gating example. (a) C code. (b) Data flow graph. (c) Schematic of generated RTL

the multiplication operation is therefore gated by condition b' so it is only executed when b' has the value 1.

Unfortunately, operation gating breaks compositionality. Recall from Section 2.2 that a key optimization involved in scaling up SEC for behavioral synthesis is the utilization of *cutpoints*. Cutpoints entail pre-verification of equivalence between corresponding internal variables in the CCDFG and the RTL, which are then replaced by (equivalent) symbolic variables. However, since the output of a gated operation is only equivalent when the gating condition is satisfied, its use as a cutpoint will cause the pre-verification to report inequivalence, breaking the compositional SEC flow.

To address this issue, we develop a *relaxed checking* algorithm for compositional SEC between a CCDFG G and a circuit M that tolerates local, "irrelevant" inequivalences for individual variables. The key idea is to continue dual-rail symbolic simulation even when a local inequivalence is encountered, but keep track of these inequivalences so that we can check if they are irrelevant during subsequent symbolic simulation. Algorithm 1 provides a high-level presentation of our approach. Here t_k denotes the scheduling step in clock cycle k, $EMap$ maps an operation op in CCDFG to combinational node in M, and x_k, s_k, i_k denote CCDFG state, circuit state, and inputs in clock cycle k respectively. At any point, the algorithm maintains a set, called $InEqSet$, of currently encountered variable inequivalences. For our example, in Fig. 1, $InEqSet$ will record the inequivalent pairs $\langle c, c' \rangle$ and $\langle d, d' \rangle$ between the CCDFG and the RTL when simulating Lines 3 and 4 respectively. During subsequent symbolic simulation, whenever an equivalence is discovered between variables in G and M, we check if that makes any of the inequivalences currently in $InEqSet$ irrelevant. For instance, when simulating Line 5 we find that e and e' are equivalent irrespective of the inequivalences between $\langle c, c' \rangle$ and $\langle d, d' \rangle$, making these two inequivalences irrelevant. When symbolic simulation terminates, one of two outcomes is possible.

- $InEqSet$ is empty, meaning all inequivalences encountered have been resolved (*i.e.*, found irrelevant). The algorithm then reports G and M to be equivalent.
- $InEqSet$ still contains some inequivalences. This means that some operations found inequivalent during symbolic simulation remain relevant even after fix-point is reached. Thus the algorithm returns G and M to be inequivalent (and outputs the unresolved inequivalences).

Algorithm 1: RELAXED-CHECKING(G, M)

```
1  k ← 0                                          ▷ Set clock cycle to 0
2  InEqSet ← ∅                                    ▷ Empty inequivalence set
3  G_Info ← FIND-GATING-INFO(G)
4  while not (checking bound or fix-point reached) do
5  |  x_{k+1} ← SIM-CCDFG(G, t_k, x_k, i_k)
6  |  s_{k+1} ← SIM-RTL(M, s_k, EMap(i_k))
7  |  foreach op_g ∈ t_k do
8  |  |  op_m ← EMap(op_g)                        ▷ find the op in circuit M
9  |  |  if not IS-EQUAL(op_g, op_m) then        ▷ SMT query
10 |  |  |  InEqSet ← InEqSet ∪ {⟨op_g, op_m⟩}
11 |  |  else
12 |  |  |  RESOLVE-INEQ(InEqSet, G_Info, op_g, op_m)
13 |  k ← k + 1
14 if |InEqSet| = 0 then                          ▷ All inequivalences resolved
15 |  return true
16 else
17 |  print InEqSet                               ▷ Report all inequivalences
18 |  return false
```

Algorithm 1 makes use of two key subroutines, FIND-GATING-INFO and RESOLVE-INEQ to do the analysis of irrelevance of local inequivalences. To describe these subroutines we first need a key definition below. For this definition, recall that a Data Flow Graph (DFG) is a directed graph $G_D = (V, E)$, where each $v \in V$ is a variable in the program, each edge $(x, y) \in E$ represents a data dependency, meaning the value of variable y depends on the value of variable x. Furthermore, we will assume that each node in G_D is labeled with an operation (*e.g.*, `add`, `mul`, etc.).[1]

DEFINITION 1 (POST DOMINANCE). *Let G_D be a Data Flow Graph for a design, and u and v be two variables. We say that u is* post-dominated *by v in G_D iff $u \neq v$ and any path that starts from u goes through v.*

REMARK 1. *Post-dominance is a common concept in compiler literature [9], although it is typically defined with respect to the Control Flow Graph instead of the DFG as above. The definition extends to a CCDFG G by taking G_D to be the DFG component of G. Given a variable mapping $EMap$, we can also extend the notion to the circuit M: a variable u' in the circuit is* post-dominated *by v' if and only if (1) there are variables u and v in G that are mapped to u' and v' respectively, and (2) u is* post-dominated *by v. Thus we will often call $\langle u, u' \rangle$ to be* post-dominated *by $\langle v, v' \rangle$.*

[1]This assumption is valid in our case since the instructions in a CCDFG are in static single assignment (SSA) form; thus each variable can be uniquely associated with one operation.

The definition of post dominance guarantees that every path from u in G_D must go through v, *e.g.*, in the example in Fig. 1(b), the variables c and d are post-dominated by e. Let $\langle u, u' \rangle$ be post-dominated by variables $\langle v, v' \rangle$ in G and M respectively. Then if v and v' are equivalent, it follows that from the perspective of any pair of corresponding variables $\langle x, x' \rangle$ that are descendants of $\langle v, v' \rangle$, the equivalence or inequivalence of $\langle u, u' \rangle$ does not matter. For instance, in Fig. 1, if e and e' are equivalent, then the inequivalence of c and c' is irrelevant. This observation leads to the theorem below that is an easy consequence of data flow.

THEOREM 1. *Suppose G is a CCDFG and M is a circuit such that the following hold: (1) variables $\langle v, v' \rangle$ are equivalent in G and M, and (2) $\langle u, u' \rangle$ are post-dominated by $\langle v, v' \rangle$ respectively. Let $\langle x, x' \rangle$ be arbitrary corresponding descendants of $\langle v, v' \rangle$. Then the equivalence between u and u' is irrelevant to the equivalence of x and x'.*

We now discuss the two subroutines.

FIND-GATING-INFO. This subroutine finds the potential gating information for a CCDFG G. A *potential gating information* is a list of pairs $\langle v, U \rangle$ where v is a variable and U is a set of variables such that each variable $u \in U$ is post-dominated by v. Theorem 1 guarantees that if v is equivalent to v' in G and M then the inequivalences of variables in U are irrelevant. Our implementation exploits the underlying LLVM constructs and information from the synthesis to efficiently determine relevant post dominance information. In particular, LLVM has a special `select` instruction of the form `y = select cond x1 x2`; the synthesis tool typically targets the condition variable of `select` instructions for operation gating.[2] Function FIND-GATING-INFO crawls over the data flow graph of CCDFG G, first identifying each `select` instruction; for each `y` it then finds all variables that are post-dominated by `y` recursively.

RESOLVE-INEQ. This function tries to resolve inequivalences in $InEqSet$ using the gating information found by FIND-GATING-INFO. Let $\langle v, v' \rangle$ be determined to be equivalent during symbolic simulation. Then we find the set U such that $\langle v, U \rangle$ is a pair computed by FIND-GATING-INFO. From the above discussion, inequivalences involving variables in U are irrelevant, therefore dropped from $InEqSet$.

3.2 Global Variables

Modular design provides several advantages by breaking the design into modules. One key optimization presented in previous work [11] is modular analysis. The basic idea is to check each module individually in a bottom up manner.

- For each module M, check the equivalence of CCDFG and RTL.
- When checking module M' that calls M, replace the invocation of M in both CCDFG and RTL by equivalent uninterpreted functions.

However, global variable usages break this modular view, and one must account for side effects on these variables while performing modular analysis. Note that while the side effects are implicit for high-level design descriptions (and hence CCDFGs), they are explicit on the synthesized RTL since the synthesis tool usually places the global variable on the interface when generating RTL.

Our solution is to compute an *extended signature* for a module that accounts for globals explicitly. Algorithm 2 shows how to compute the extended signature of a module. The key idea is to analyze the module to determine the globals used in the module. The parameters of the module are then extended to include *read-only* and *read-write* globals among the inputs and *write-only* and *read-write* globals among the outputs.[3]

Algorithm 2: GET-EXTENDED-SIGNATURE(f)

```
1  I ← PARAMETERS(f)
2  O ← OUTPUTS(f)
3  V_G ← FIND-ALL-GLOBALS(f)
4  foreach v ∈ V_G do
5      switch USAGE-TYPE(v) do
6          case R :  I ← I ∪ {v} ;          ▷ read-only
7          case W :  O ← O ∪ {v} ;          ▷ write-only
8          case RW :                        ▷ read-and-write
9              I ← I ∪ {v}
10             O ← O ∪ {v};
11 return ⟨I, O⟩
```

```
1  char A;       // global variable
2  int B;        // global variable
3  int C[2];     // global variable
4  void top() {
5    int i = 10;
6    bar(i);
7  }
8  void bar(int x) {
9    B = A + x;          // side effect
10   C[1] = C[0] + x;    // side effect
11 }
```

Figure 2: Global variable usage example

Fig. 2 shows an example of the computation. Module `top` includes `bar` as a sub-module which updates the global array `C`. Based on the parameters, the signature of `bar` is: `bar :: int -> void`. However, accounting for globals, the extended signature of `bar` by using Algorithm 2 is the following,

```
bar ::  int->char->int[2] -> (int, int[2])
```

meaning that `bar` is represented as a function of three inputs (of the specified types), generating a pair of outputs.

Extended signatures account for global variables during modular analysis. Suppose that `bar` has been certified; when certifying `top` we replace `bar` with an uninterpreted function (say `BAR`) of three arguments, and the effect of the invocation of `bar` on the globals is given by $(\texttt{B}, \texttt{C}) = \texttt{BAR}(\texttt{i}, \texttt{A}, \texttt{C})$.

[2] U need not be the *complete* set of variables post-dominated by v. This permits us to merely consider conditions in the LLVM `select` instruction as potential gating information. This runs the risk of possible spurious SEC failures. However, in our experience, this check has been sufficient.

[3] The algorithm assumes that local variables within a module have been standardized apart via renaming to avoid name conflicts with the globals and consequent variable capture. In our case capture avoidance is trivial since LLVM adopts different naming conventions for local and global variables.

Table 1: Summary of Evaluation on CHStone Benchmark

App. Domain	Design	Lines of code		C	RTL	Operation	Global Variables[a]			Time	Mem.
		C	RTL	Functions	Modules	Gating	R	W	RW	(s)	(MB)
Arithmetic	DFADD	526	3722	17	5	Yes	4	0	1	174.9	169.34
	DFDIV	436	5192	19	4	Yes	4	0	1	6946.1	594.87
	DFMUL	376	3115	16	2	Yes	4	0	1	63.5	75.31
	DFSIN	755	11224	31	8	Yes	6	0	1	7151.3	603.50
Microprocessor	MIPS	232	2944	1	1	No	1	0	0	250.4	125.21
Media Processing	ADPCM	541	14935	15	5	No	15	19	63	68.2	105.45
	GSM	393	5598	12	4	Yes	4	0	0	49.6	83.07
	JPEG	1692	32846	30	17	Yes	30	14	17	2187.3	375.90
	MOTION	583	6168	13	5	Yes	9	0	4	1515.1	408.77
Security	AES	716	11869	11	7	Yes	4	0	5	170.7	106.59
	BLOWFISH	1406	17420	6	4	No	3	0	4	44.9	91.89
	SHA	1284	18819	8	4	No	3	0	4	6.0	89.04

[a]R means read-only, W means write-only, and RW means read-and-write.

4. EXPERIMENTAL RESULTS

4.1 Performance Evaluation

We have applied our framework to certify synthesized RTL for all the ESL designs in the CHStone benchmark. CHStone is a publicly available benchmark suite for behavioral synthesis, that includes twelve designs selected from different application domains. We used a commercial behavioral synthesis tool to synthesize the RTL. The most complex design in the benchmark is `JPEG` which has more than 32K lines of RTL code. For our experiments we have used the benchmark designs as is with one modification: two designs, `JPEG` and `MOTION`, used double pointers to represent two-dimensional arrays; these were modified to eliminate the double-pointer and represent the arrays explicitly. The reason has to do with the quirks of the synthesis tool used in this experiment. The synthesis tool inlines functions that have double pointers, thus flattening the module structure in the synthesized RTL. In addition to generating significantly larger RTL, this also destroys the module structure in the synthesized design. Since scalability of *modular analysis* (in the presence of design optimizations) is the key target of the experiments, we found the original designs unsuitable as targets for evaluation. The experiments were conducted on a workstation with 3GHz Intel Xeon processor and 8GB memory. For each design, we checked the equivalence between its CCDFG and RTL via dual-rail symbolic simulation, which symbolically simulates the CCDFG and RTL clock cycle by clock cycle. After each clock cycle, we checked the equality of mapped variables in the CCDFG and RTL by the MathSAT SMT solver [6]. We also applied cutpoints, cut-loop, and modular analysis optimizations when checking each design.

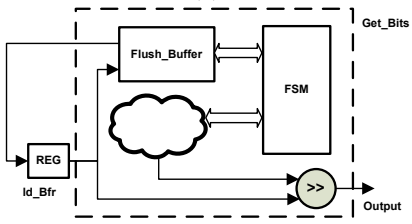
Table 1 shows the results of the experiments. The `JPEG` design takes about 36 minutes with 375.9 MB memory usage. The maximum certification time is required for `DFSIN`, which takes around 119 minutes with 603.5 MB memory usage. The experiment results demonstrate that independent of application domain our framework scales up to designs of practical complexity. No other SEC framework to our knowledge can handle behaviorally synthesized designs at this scale. Furthermore, only `MIPS` can be certified without handling operation gating and global variable optimizations.

4.2 A Behavioral Synthesis Bug

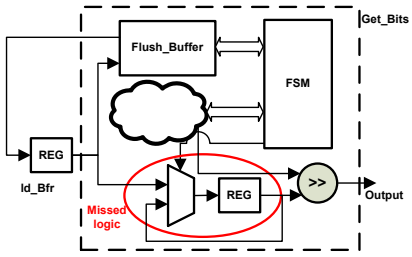
Our experiments found a bug in the synthesis tool during the certification of the `MOTION` design, which is a C implementation of a motion vector decoding algorithm for MPEG-2. Fig. 3(a) shows the source code fragment that triggers the bug. Here `ld_Bfr` is a global variable. In function `Get_Bits`, the return value `Val` is computed by right-shifting `ld_Bfr`. After `Val` is computed, `ld_Bfr` is updated in the subroutine `Flush_Buffer`. The update performed by

```
1  unsigned int ld_Bfr; // global ld_Bfr
2  void Flush_Buffer(int N) {
3    // modify the global variable
4    ld_Bfr = update(N, ld_Bfr);
5  }
6  unsigned int Get_Bits(int N){
7    unsigned int Val;
8    Val = ld_Bfr >> (32 - N);
9    Flush_Buffer(N);
10   return Val;
11 }
```

(a)



(b)



(c)

Figure 3: Bug found in MOTION example. (a) C source code (b) Wrong RTL (c) Correct RTL

`Flush_Buffer` does not affect the return value. Fig. 3(b) shows the RTL implementation synthesized by the behavioral synthesis tool. The global variable `ld_Bfr` is synthesized to a register outside of module `Get_Bits`. The output of `Get_Bits` is thus a combinational circuit with `ld_Bfr` as input. Therefore, when sub-module `Flush_Buffer` produces a new data for `ld_Bfr`, the new data is propagated to the output in the same clock cycle, leading to a wrong output.

The bug is caused because behavioral synthesis applies aggressive transformations to minimize resource usage. As can be seen by comparing Figs. 3(b) and 3(c), the synthesis tool in this case eliminates a register without correctly taking into account the side effect on the global variable. Such subtleties reinforce the need for SEC for certification of synthesized RTL designs. The bug has been confirmed by developers of the synthesis tool and fixed in a new release.

5. RELATED WORK

Koebl *et al.* [15] provides a good overview of research in SEC between high-level and RTL designs. Recently increasing sophistication of behavioral synthesis has resulted in several SEC optimizations to scale up certification of synthesized RTL [13, 16, 14, 11, 10]. For instance, Vasudevan *et al.* [20] introduce *sequential compare points* as a set of observable signals to be compared between high-level designs and RTL. There are commercial tools [1, 14] that can apply SEC between RTL and high-level (C/C++/SystemC) models. However, we have found no published results on approaches to handle design and implementation optimizations in any certification framework for behavioral synthesis.

There has however been research on handling such optimizations in SEC comparing RTL and netlist designs. Baumgartner *et al.* [5] discuss an approach for invariant generation to address the conditional equivalence checking problem for optimizations including clock gating and power gating. Moon *et al.* [18] propose equivalence checking techniques that exploit well-partitioned circuit structures.

6. CONCLUSION AND FUTURE WORK

We have presented a SEC framework to compare an ESL design with its behaviorally synthesized RTL implementation in the presence of design and implementation optimizations, *e.g.*, operation gating and global design variables. The framework scales to practical designs: it can handle all designs of the CHStone benchmark, some of which have more than 32K LoC synthesized RTL. We do not know of any other tool that can handle diverse designs at this scale, and the algorithms presented here are crucial to this scalability. In addition, certification found a bug in a commercial synthesis tool, underlining the importance of SEC for behavioral synthesis and the effectiveness of our framework. In future work, we plan to develop techniques for handling more sophisticated designs, including concurrent ESL specifications and complex, synthesized module interfaces.

7. ACKNOWLEDGMENTS

This research was partially supported by National Science Foundation Grants #CCF-0916772 and #CCF-0917188 and by a research grant from Intel Corporation. We thank Disha Puri, Naren Narasimhan, and Jin Yang for advice and help.

8. REFERENCES

[1] *Sequential Equivalence Checking: A new approach to functional verification of datapath and control logic changes*, 2007.

[2] *C-to-Silicon Compiler User Guide, 11.10*, 2011.

[3] *Catapult C Reference Manual*, 2011.

[4] *Synthesizer Reference Guide, 4.1*, 2011.

[5] J. Baumgartner, H. Mony, M. L. Case, J. Sawada, and K. Yorav. Scalable conditional equivalence checking: An automated invariant-generation based approach. In *FMCAD*, 2009.

[6] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT solver. In *CAV*, 2008.

[7] J. Cong, B. Liu, R. Majumdar, and Z. Zhang. Behavior-level observability analysis for operation gating in low-power behavioral synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 16(1), 2010.

[8] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for FPGAs: from prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, April 2011.

[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3), July 1987.

[10] K. Hao, S. Ray, and F. Xie. Equivalence checking for behaviorally synthesized pipelines. In *DAC*, 2012.

[11] K. Hao, F. Xie, S. Ray, and J. Yang. Optimizing equivalence checking for behavioral synthesis. In *DATE*, 2010.

[12] 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. *Journal of Information Processing*, 17, 2009.

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

[14] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley. Solver technology for system-level to RTL equivalence checking. In *DATE*, 2009.

[15] A. Koelbl, Y. Lu, and A. Mathur. Embedded tutorial: formal equivalence checking between system-level models and RTL. In *ICCAD*, 2005.

[16] S. Kundu, S. Lerner, and R. Gupta. Validating high-level synthesis. In *CAV*, 2008.

[17] The LLVM Compiler Infrastructure. *LLVM Language Reference Manual (Version 2.7)*, 2010.

[18] I.-H. Moon, P. Bjesse, and C. Pixley. A compositional approach to the combination of combinational and sequential equivalence checking of circuits without known reset states. In *DATE*, 2007.

[19] S. Ray, K. Hao, F. Xie, and J. Yang. Formal verification for high-assurance behavioral synthesis. In *ATVA*, 2009.

[20] S. Vasudevan, J. Abraham, V. Viswanath, and J. Tu. Automatic decomposition for sequential equivalence checking of system level and RTL descriptions. In *MEMOCODE*, 2006.