Scalable Certification Framework for Behavioral Synthesis Front-End

Zhenkun Yang[1], Kecheng Hao[1], Kai Cong[1], Li Lei[1], Sandip Ray[2] and Fei Xie[1]
[1]Dept. of Computer Science, Portland State University, Portland, OR 97207, USA
{zhenkun, kecheng, congkai, leil, xie}@cs.pdx.edu
[2]Strategic CAD Labs, Intel Corporation, Hillsboro, OR 97124, USA
sandip.ray@intel.com

ABSTRACT

Behavioral synthesis entails application of a sequence of transformations to compile a high-level description of a hardware design (e.g., in C/C++/SystemC) into a register-transfer level (RTL) implementation. In this paper, we present a scalable equivalence checking framework to validate the correctness of compiler transformations employed by behavioral synthesis front-end. Our approach makes use of dual-rail symbolic simulation of the input and output of a transformation, together with identification and inductive verification of their loop structures. We have evaluated our framework on transformations applied by an open source behavioral synthesis tool to designs from the CHStone benchmark. Our tool can automatically validate more than 75 percent of the total of 1008 compiler transformations applied, taking an average time of 1.5 seconds per transformation.

1. INTRODUCTION

Behavioral synthesis is the process of compiling an Electronic System Level (ESL) design to register-transfer level (RTL). ESL specifications define the design functionality at a high level of abstraction (*e.g.*, with C/C++ or SystemC), and thus provide a promising approach to address the exacting demands to develop feature-rich, optimized, and complex hardware systems within aggressive time-to-market schedules. However, the adoption of the approach critically depends on our ability to ensure that the synthesized design indeed correctly implements the ESL specifications.

A behavioral synthesis flow can be roughly divided into two phases: *front-end* and *back-end*. The front-end primarily entails compiler transformations; the goal is to reduce code complexity of the generated design, maximize data locality, etc. [4], and transform the design into a form more suitable for resource allocation and control synthesis. The back-end entails local, sometimes manual, optimizations for a number of metrics, *e.g.*, performance, power consumption, etc.

Previous work [8,20] developed a scalable sequential equivalence checking (SEC) tool for behavioral synthesis back-end. The tool performs dual-rail symbolic simulation between RTL and the internal representation (IR) extracted after the compiler and scheduling transformations have been applied. The key observation is that SEC can exploit the operation-to-resource mappings employed by the synthesis tool: these mappings provide a liberal source of internal signals for use as cutpoints. The tool could certify synthesized hardware designs with tens of thousands of lines of RTL.

However, the above framework is meaningful only if front-end transformations are also certified. Front-end transformations constitute the majority of synthesis transformations *e.g.*, more than a hundred such transformations are applied to a design. Furthermore, they are applied aggressively under delicate, implicit invariants, and tend to be complex and error-prone. Unfortunately, the SEC approach for the back-end is ineffective for front-end transformations. Back-end SEC relies on operation-to-resource mappings to correlate internal signals of IR and RTL; such mappings are not available between IRs during front-end transformations. Consequently, a naïve SEC approach would require cost-prohibitive input-output equivalence checking of different IRs. Recent work [19] developed a technique for certifying compiler transformations by symbolically exploring corresponding paths of the IRs before and after each front-end transformation, but this approach was also found prohibitive for most practical designs, *e.g.*, designs with loop structures.

In this paper, we develop a tool for certifying front-end transformations. As in previous work [19], our approach entails symbolic execution of IRs. However, we augment it with optimizations to handle path explosion. Our tool has been used to certify compiler transformations applied by the LegUp behavioral synthesis tool [3] on the CHStone benchmark [9]. CHStone is a publicly available C-based ESL benchmark suite; some designs have over 1200 lines of C, and generate over $53,000$ lines of RTL. Our tool can automatically certify more than 75% of the 1008 compiler transformations involved. The suite includes designs with complex branching structures and unbounded loops. We are not aware of any other SEC framework that can certify front-end compiler transformations of such diversity and scale.

Our optimizations include a compositional strategy for containing path explosion, and an inductive assertions approach for reasoning about loops. While the strategies are well-known, their application to front-end transformations is a challenge. For example, the inductive assertions strategy entails "cutting the loop" for the IRs, reducing the fixpoint computation to checks at the entry, body, and exit. The idea is derived from classic works in program verification [5, 10],

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
DAC '14 June 01 - 05 2014, San Francisco, CA, USA
Copyright 2014 ACM 978-1-4503-2730-5/14/06 ...$15.00.

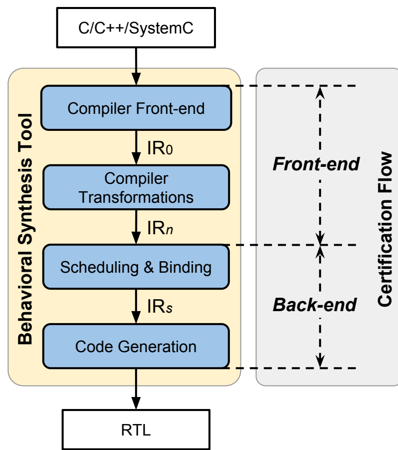

Figure 1: Behavioral synthesis and certification flow

and has been applied before for the back-end [8]. However, it requires identification of corresponding loop structures of two IRs and corresponding variables in the IRs affected by loop iterations. Identifying such variables is easy for back-end by operation-to-resource mappings. However, in the absence of such mappings, we need more elaborate algorithms and heuristics for enabling cut-loop on front-end. Our key contribution is to integrate such strategies by exploiting information from the synthesis tool.

2. BACKGROUND

2.1 Behavioral Synthesis

Figure 1 shows the overall working of a behavioral synthesis tool. Similar to a generic compiler, it first performs lexical, syntax and semantic analysis, and builds an IR of the ESL description. The IR is then subjected to a sequence of transformations which can be categorized into phases.

1. *Front-end transformations* include generic compiler optimizations, *e.g.*, constant propagation, loop unrolling, code motion, etc.

2. *Back-end* involves scheduling the operations of the design to specific clock cycles, binding operations to functional units, allocating resources, synthesis of control circuit to implement schedule, and generating RTL. Further manual tweaks are also implemented to optimize for timing, power, etc.

2.2 Back-end Certification Framework

Previous work [8, 16, 20] developed an SEC framework to certify RTL designs generated by behavioral synthesis. The key observation is that SEC via dual-rail symbolic simulation is effective for comparing RTL with the IR obtained after scheduling and binding transformations (*viz.*, IR_s). Recent work [7] extended it to scheduling transformations, *e.g.*, loop and function pipelining.

2.3 Front-end Certification and Complexity

Unfortunately, a direct dual-rail SEC does not work for front-end. The dissimilarity in structure between the input ESL description (IR_0 and the IR IR_n generated after front-end transformations) makes it difficult to find direct mappings between internal variables. The back-end exploits

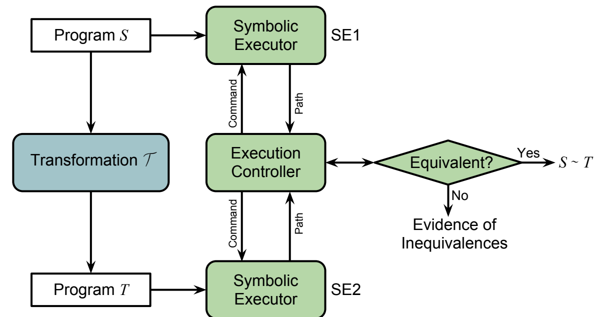

Figure 2: Framework of checking equivalence between program S and T, which are the input and output of transformation $\mathcal{T}$ respectively.

operation-resource mappings of the synthesis tool to identify cutpoints; such mappings are not available for front-end. Finally, transformation implementations are typically closed-source (in addition to being highly complex), which precludes certifying the implementations by code analysis.

Recent work [19] extended the certification framework to front-end by decomposing it into a series of checks, one for each transformation applied. The key observation is that while the transformation implementations may be closed-source, it is possible to obtain from most tools the IRs after application of each front-end transformation. Thus, an SEC methodology was developed to compare each pair of consecutive IRs, as shown in Figure 2. The idea is to symbolically execute the two IRs (referred to as S and T) and check that each pair of corresponding program paths is equivalent. A program path is uniquely specified by the sequence of branch conditions that must hold for the control flow to execute the instructions in the path. The approach was used on some cryptographic applications. Unfortunately, the approach does not scale to other practical programs. In particular, it requires effective enumeration of all paths in S and T; in practice, this can lead to *path explosion*. Two key sources of path explosion in practice are subroutine calls and loops. In the next two sections we discuss optimizations to address these problems.

3. MODULAR REASONING ACROSS FUNCTIONS

Why do subroutine calls contribute to path explosion? The naïve approach of symbolically executing the program treats each function as if it were inlined: the function body is symbolically executed at each call site. When a function f having a number of branches in its body (and hence program paths) is invoked many times, each invocation contributes a multiplicative factor to the number of paths explored.

Our approach to address this problem is to develop a compositional approach to symbolic execution [6], which permits equivalence checking on a per-function basis. Suppose functions f and f' invoke functions g and g' respectively. Then (1) we separately check the equivalence of g and g'; and (2) when checking the equivalence of f and f', we replace g and g' with the same uninterpreted function symbols.

Of course the above naïve scheme only works for *side effect* free functions. If g and g' update some global variables or pass-by-reference arguments, then replacing g and g' with the same uninterpreted function on the explicit arguments will be unsound since the effect on the global variable or pass-by-reference arguments is not accounted for. To ad-

dress this, we use a notion of "extended signature". The idea is to extend the type signature of a function explicitly accounting for the side effects. Let g be a sub-function; we use $\tau = g(\vec{\alpha_v}, \vec{\alpha_r})$ to represent the signature of g, where τ denotes the return value, $\vec{\alpha_v}$ denotes pass-by-value arguments, and $\vec{\alpha_r}$ denotes pass-by-reference arguments. Then, in addition to the function arguments, suppose function g reads globals $\vec{\beta_r}$ and updates globals $\vec{\beta_w}$. The extended type signature of g is:

$$\langle \tau, \vec{\alpha_r'}, \vec{\beta_w'} \rangle = g(\vec{\alpha_v}, \vec{\alpha_r}, \vec{\beta_r}),$$

where $\vec{\alpha_r'}$ and $\vec{\beta_w'}$ are updated versions of $\vec{\alpha_r}$ and $\vec{\beta_w}$ respectively, mimicking the notion that they may be arbitrarily changed by function g. [1]

```
1   char A;                // global variable
2   int B;                 // global variable
3   int C[2];              // global variable
4   void f(int d[4]) {
5     int i = 8;
6     g(i, d);
7   }
8   void g(int x, int y[4]) {
9     B = A + x;           // side effect on global B
10    C[1] = C[0] + x;     // side effect on global C
11    y[1] = y[0] + x;     // side effect on arguments
12  }
```

Figure 3: Global variable usage with sub-function call example.

Figure 3 shows an example of a sub-function with side effects. Function `f` invokes function `g` which updates the globals `B` and `C`, and pass-by-reference argument `y`.

Extended signatures are exploited to replace function calls with uninterpreted functions symbols. Suppose that function `g` has been certified; when certifying function `f`, we replace `g` with an *uninterpreted function* (say $\mathcal{G}$) of four arguments, and the effect of the invocation of `g` on the globals (`B` and `C`) and argument `y` is given by:

$$\langle \mathtt{d}, \mathtt{B}, \mathtt{C} \rangle = \mathcal{G}(\mathtt{i}, \mathtt{d}, \mathtt{A}, \mathtt{C}).$$

Since each invocation of sub-functions is replaced by an uninterpreted function symbol, we alleviate path-explosion problem introduced by subroutine calls. It is worth noting that this can generate false alarms, and we need to take special care to handle common false negatives. We discuss this issue when describing our experiments.

4. HANDLING LOOPS

Loops are the second major contributors to path explosion (and, in case of unbounded ones, non-termination) in symbolic simulation of software programs. The reason is that symbolic simulation of a loop induces (at least) two branches for each loop iteration simulated: (1) the branch where the loop test holds (and hence the body is executed) and (2) the branch where the test is false. [2]

[1] Pass-by-reference arguments are pointers, and most behavioral synthesis tools restrict the usage of pointers to compile-time determinable ones, which makes this approach works in finding which variable a pointer points to.

[2] The branching may be limited if the value of the loop test can be computed concretely during symbolic execution of the loop. However, this is not possible for most non-trivial loops in practice.

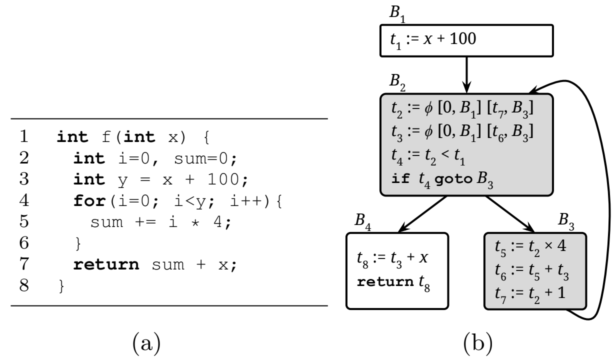

Figure 4: A simple function with a loop in C and its IR. (a) Function `f` has an unbounded `for` loop. (b) The IR of `f`, with boxes representing basic blocks, and arrows representing control flow. Control flow merge is implemented via ϕ-instructions in basic block B_2.

We handle equivalence of loops by an approach called *cut-loop optimization*. Our approach borrows ideas from a corresponding one for back-end SEC between high-level IR and RTL [8], and is an adaptation of classic inductive assertions approach [5, 10] to program equivalences. The idea is to "cut" the loop, which reduces equivalence of loop execution to equivalence checks at entry, body, and exit. To illustrate the idea, consider the example in Figure 4. Figure 4 (a) shows a simple unbounded loop in C. Figure 4 (b) shows the IR of function `f`. [3] If the input `x` is symbolic, then the symbolic expansion of the loop (and hence the symbolic execution of `f`) will not terminate. [4] We assume loops are in *natural loop* form in the IRs. A *natural loop* must have a single entry point (header) and at least one *back edge* leading the control flow from the loop body back to the loop header. Figure 4 (b) is an example of a natural loop, where B_2 is the loop header, and edge (B_3, B_2) is the back edge.

As pointed out in Section 2.3, back-end SEC [8] exploits mappings of variables provided by behavioral synthesis tool. For simplicity, suppose each basic block is a scheduling step in Figure 4 (b). At the end of execution of each scheduling step, we check the equivalence of all mapped variables. If they are equivalent, we replace every pair of mapped variables with the same symbolic symbol (with *cut-point* optimization). Suppose the loop is entered from B_1, after executing of B_3, variables t_5, t_6 and t_7 are checked equivalence with their mapped variables in RTL, and then will be replaced with symbolic variables. Till this point, we checked the first iteration of the loop. The subsequent iterations of the loop are all led by the back edge. Since we made t_5, t_6 and t_7 symbolic, executing the subsequent iterations once will cover all possible cases. All we need to do is to avoid multiple entrance of a loop through the same back edge.

However, variables mappings between two IRs are not available during front-end transformations. Suppose a loop consists of a list of basic blocks; then we must identify a

[3] Control flow merge is implemented via ϕ-instructions in basic block B_2. A ϕ-instruction $\mathtt{v} = \phi\,[\alpha, B_i]\,[\beta, B_j]$ in basic block B means that `v` has the value α if B is reached from B_i, and β if reached from B_j.

[4] Technically, symbolic simulation can terminate when a fixpoint is reached, *i.e.*, when all reachable states have been explored. But achieving such fixpoint requires the restriction that all the variable types are finite, as well as an expensive fixpoint computation through symbolic simulation.

minimum set of variables that need to be mapped between the two IRs. In particular, we need to identify mappings for loop-carried variables. The reason is that we want to make them symbolic, so that we only need to execute the loop once through each back edge (see below).

We achieve the above through *use-definition chains* analysis [1]. In the example shown in Figure 4 (b), loop L consists of basic blocks B_2 and B_3, variable t_6 and t_7 are loop-carried variables. For example, execution of loop L is done as follows:

1. Loop L is entered from B_1:
   - path 1: $B_1 \rightarrow B_2 \rightarrow B_4$: we check the equivalence of return variable t_8;
   - path 2: $B_1 \rightarrow B_2 \rightarrow B_3$: we check the equivalence of loop-carried variables t_6 and t_7, and make them symbolic afterwards.
2. Loop L is entered from B_3 through the back edge:
   - path 1: $B_3 \rightarrow B_2 \rightarrow B_4$: we check the equivalence of return variable t_8;
   - path 2: $B_3 \rightarrow B_2 \rightarrow B_3$: we check the equivalence of loop-carried variables t_6 and t_7, and terminate L.

The above approach requires detecting the loop structure (*e.g.*, loop header, exit, back edge, etc.) in the IRs. Once the loop-carried variables are identified, equivalence of loop computation can be verified by checking the first iteration (entered from entry) and one subsequent symbolic iteration (entered through back edge) for each back edge; since we made loop-carried variables symbolic after first iteration, this covers all possible cases for the subsequent iterations. The sufficiency of these checks was mechanically proven in previous work using the ACL2 theorem prover [17].

Finally, cut-loop requires that when the corresponding loops in the two IRs being compared have the same structure, and perform equivalent computation at each iteration, *e.g.*, it is inapplicable if the transformation entails partial loop unrolling. However, as our experiments indicate, most behavioral synthesis transformations are structure-preserving, making the optimization widely applicable.

5. EXPERIMENTAL RESULTS

We applied our framework to CHStone [9], a publicly available behavioral synthesis benchmark suite containing 12 ESL designs (in C). We used LegUp [3] to synthesize these designs. We conducted our experiments on a workstation with Debian 7.1 running on a 2.93 GHz Intel Xeon X3470 processor with 8 GB of memory. We focused on intra-procedural transformations. The experiments were run with a cutoff time of 90 seconds: certifications taking longer than this time are classified as failures. The reason for this cutoff is that in our experience, most successful transformation certifications that complete in any reasonable time finish within a few seconds for examples of this size; if symbolic execution takes more than 90 seconds, it is unlikely to finish. Thus we believe that the impact of making the cutoff longer on the number of successful transformations will be insignificant. Our tool supports a number of SMT solvers. The results on this benchmark use Z3 since it outperforms others.

Table 1 shows the statistics of the experiments, *e.g.*, the number of transformations applied by the synthesis tool,[5]

[5]Some transformations are applied more than once.

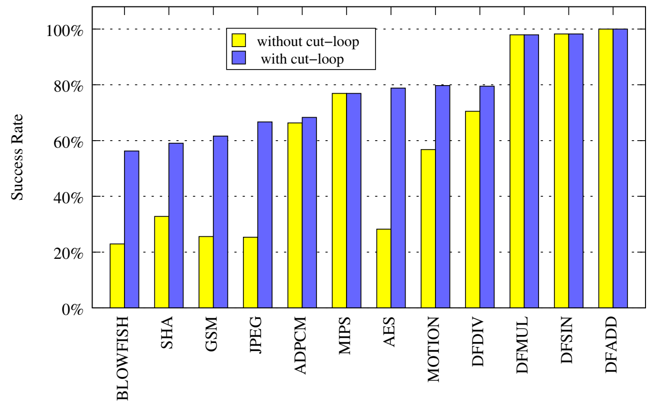

Figure 5: Comparison of success rate on designs of CHStone benchmark without and with cut-loop optimization. The x axis is ordered by the success rate with cut-loop.

the number of transformations checked successfully, as well as time and memory usages. In all successful cases, time and memory usages are modest. With compositional execution and cut-loop optimization, we successfully validated 75.69 percent of transformations (763 out of 1008).

Figure 5 compares the success without and with cut-loop optimization. Without cut-loop, we can validate only 52.88 percent of transformations (533 out of 1008). Thus cut-loop provides an improvement of 22 percent. The improvement is most significant for `AES`, `JPEG`, `GSM`, and `BLOWFISH` since they have more loops. The transformations that fail certification (about 25 percent) typically do so for two reasons: (1) the transformation changes loop structure making cut-loop inapplicable; or (2) symbolic expressions for corresponding variables in source and target programs may become too complex, causing blow-up for the SMT solver.

Since we focus on intra-procedural transformations, we check the designs compositionally; this may sometimes introduce subtle false alarms. Furthermore, false alarms can arise in surprising ways, *e.g.*, if the extended signature of a sub-function is different in the source and target programs. As an example [6], suppose that a function `foo` has a sub-function `legup_memcpy` that is invoked as follows.

```
r = legup_memcpy(a, b, c)
```

This sub-function has a return value, but the value is never used subsequently. Similarly, suppose that a function `bar` invokes the same sub-function `legup_memcpy` as follows:

```
legup_memcpy(a, b, c)
```

This invocation treats `legup_memcpy` as if it returns void. The problem is when we compositionally check `foo` and `bar`, we abstract sub-function `legup_memcpy` with uninterpreted function. Since the extended signatures are different for two invocations of the sub-function, (one with a return value and the other one without), a naïve approach will report an inequivalence due to type mismatch. Since the result of return value `r` in `foo` is not used, this inequivalence is a false alarm. However, it can be easily eliminated, by excluding from the extended signature the types of return values that are not subsequently used.

[6]This is a real example in function `Fill_Buffer` in `MOTION` design. The transformation is called “Combine Redundant Instructions”

Table 1: Summary of Evaluation on CHStone Benchmark

App. Domain	Design	Lines of C Code	Lines of RTL	# of Functions	# of Checked Transformations	# of Successful Checks	Success Rate (%)	Avg. Time (s)	Memory (MB)
Arithmetic	DFADD	542	12933	17	62	62	100.00	0.78	159.86
	DFDIV	452	10948	19	62	78	79.49	0.95	161.22
	DFMUL	392	7100	16	47	48	97.92	0.93	155.93
	DFSIN	772	22949	31	113	115	98.26	0.88	187.70
Microprocessor	MIPS	256	7237	1	10	13	76.92	2.32	15.01
Media Processing	ADPCM	521	33706	15	69	101	68.32	3.33	123.85
	GSM	388	22816	12	53	86	61.63	0.26	122.94
	JPEG	1031	53584	30	158	237	66.67	1.55	694.76
	MOTION	414	13770	13	59	74	79.73	0.49	52.50
Security	AES	699	40014	11	67	85	78.82	4.22	120.56
	BLOWFISH	1241	23490	6	27	48	56.25	3.28	93.58
	SHA	1284	12491	8	36	61	59.02	0.05	106.14

6. RELATED WORK

There has been research on formally proving compiler transformations correct by theorem prover. CompCert [12] is the first formally verified compiler. Similar to CompCert, Vellvm [22] project formalizes LLVM's intermediate representation, and develops a framework for reasoning about programs. Ray et al. propose an sequential equivalence checking framework for certifying behaviorally synthesized RTL [16]. This framework uses theorem proving to certify high-level transformations. However, using theorem proving to prove all transformations requires enormous manual effort; it also requires knowledge about internal algorithm of each transformation, which is often not available because most behavioral synthesis tools are closed source.

Pnueli et al. proposed the notion of *translation validation* [14] for validating the transformations during compilation. Instead of verifying a transformation once and for all, they show how to generate a proof of correspondence between the source and target programs for each individual transformation. However, it restricts the source and target programs each consists of one single loop. Zuck et al. extended this approach to support structure-modifying transformations [23]. Necula used symbolic evaluation techniques from proof-carrying code to tackle translation validation [13]. However, this approach only handled transformations where source and target programs have the same branch conditions. Zaks and Pnueli proposed a framework to construct the cross product of the source and target program [21]. This reduces the problem of checking the equivalence of two programs to verification of a single program. Peggy [18] performs translation validation for the LLVM compiler using *equality saturation*. It builds Program Expression Graphs for the source and target programs of a transformation and then reasons about equalities among nodes. If output nodes of two programs are shown equal, the two programs are equivalent. To our knowledge, these approaches do not scale to programs of the size we consider in this paper. Note that a key reason for the difference in scalability is that the aim of the above line of research is to check the correctness of generic compiler transformations while we focus on transformations in behavioral synthesis. In particular, the programs being synthesized are finite state and many language features such as dynamic memory allocation are prohibited.

There has also been recent research on applying symbolic techniques to checking the equivalence of two arbitrary programs. UC-KLEE [15] proposes a smart stub function that invokes two routines that need to be verified for equivalence, and leverages KLEE [2] to symbolically execute the stub function. Upon finding a path, UC-KLEE checks if the two routines behave the same. UC-KLEE enumerates path with best efforts; thus it suffers from path explosion and does not terminate when executing unbounded loops. SYM-DIFF [11] is a symbolic differentiation tool, which symbolically executes two programs with same symbolic inputs, and checks if the two programs have identical outputs. SYM-DIFF handles loops by unrolling them to a user-specified depth; consequently it cannot certify equivalence between programs whose loops iterations are long or controlled by input variables; we can handle such programs with cut-loop optimization.

7. CONCLUSION AND FUTURE WORK

We have presented a scalable SEC framework to validate the correctness of front-end compiler transformations in behavioral synthesis. We use symbolic execution technique to explore (possibly all) paths of the source and target programs of each transformation. We showed how to ameliorate path explosion and non-termination in symbolic simulation through compositionality and cut-loop optimization. Our framework can fully automatically certify results of more than 75 percent of 1008 transformations employed by a synthesis tool on designs from the CHStone benchmark. We are not aware of any SEC framework that can handle compiler transformations at such diversity and scale.

Our results underline the importance of aligning verification methodology with the design flow in the development of a scalable verification framework. SEC for behavioral synthesis transformations at the scale achieved here has not been done before because extant tools focused on input/output equivalence between the high-level ESL description and synthesized RTL; such efforts are ineffective because of the high abstraction gap. On the other hand, pre-certified compiler transformation via theorem proving as proposed in previous work [16] was not successful both because of the number and complexity of such transformations and the reluctance of synthesis tool vendors to expose transformation implementation for formal analysis. Our key insights are that (1) design IRs before and after each transformation application can be made available from a commercial synthesis flow even if the transformations themselves are proprietary, and (2) restrictions in program features enforced by behavioral synthesis from the need to eventually generate hardware circuit from the design description make it possible to use "black-box" SEC techniques effectively to certify these IRs. The key take-away from our paper is that once the right verification methodology has been identified, it is possible with insight of the source of verification complexity of

the domain to adapt well-known analysis ingredients into an end-to-end certification solution in a complex domain.

One possible argument against our framework is the requirement that IRs after each transformation application be available to the tool. In particular, if the validation is performed by a third party, this requirement may provide exposure to confidential design intellectual property (IP). In practice, we have not seen that to be a problem for two reasons. First, in many industrial contexts, the validation is performed by personnel who have access to the original ESL and RTL designs anyhow (*e.g.*, by a validation group in the same organization that designed the ESL). Second, most extant commercial behavioral synthesis tools already provide the information on IRs; we do not require any additional information to perform our analysis. Nevertheless, the potential of IP leakage is an important one, and we plan to look at the constraints and data available to third-party evaluators during design certification in future work to determine how our framework can be made usable in that context.

In future work, we also plan to extend our SEC framework to handle more aggressive transformations. The fact that we still cannot certify 25 percent of the transformations in CHStone shows that there is significant room for improvement. Our planned future extensions include equivalence checking for transformations spanning multiple procedures or functions. We are also planning to handle transformations that modify structures of loops, perhaps through domain-specific SEC optimizations. Recall that a key reason for our inability to handle the transformations where SEC fails is the inapplicability of cut-loop, which requires equivalence for each iteration of corresponding loops of the two programs. We are looking for ways to loosen that restriction so that transformations such as partial loop unrolling can be certified. Last but not least, we are also looking at certifying scheduling and resource binding operations to complete the certification flow.

8. REFERENCES

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

[2] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of OSDI*, pages 209–224, 2008.

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. Legup: high-level synthesis for FPGA-based processor/accelerator systems. In *Proc. of FPGA*, pages 33–36, 2011.

[4] 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.

[5] R. Floyd. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematcs*, volume XIX, pages 19–32. American Mathematical Society, 1967.

[6] P. Godefroid. Compositional dynamic test generation. In *Proc. of POPL*, pages 47–54. ACM, 2007.

[7] K. Hao, S. Ray, and F. Xie. Equivalence checking for behaviorally synthesized pipelines. In *Proc. of DAC*, pages 344–349, 2012.

[8] K. Hao, F. Xie, S. Ray, and J. Yang. Optimizing equivalence checking for behavioral synthesis. In *Proc. of DATE*, pages 1500–1505, 2010.

[9] 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*, 4(4):740–752, 2009.

[10] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, 1969.

[11] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: a language-agnostic semantic diff tool for imperative programs. In *Proc. of CAV*, pages 712–717, 2012.

[12] X. Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, Dec. 2009.

[13] G. C. Necula. Translation validation for an optimizing compiler. In *Proc. of the ACM SIGPLAN on PLDI*, pages 83–94, 2000.

[14] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proc. of TACAS*, pages 151–166, 1998.

[15] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *Proc. of CAV*, pages 669–685, 2011.

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

[17] S. Ray, W. A. Hunt, Jr., J. Matthews, and J. S. Moore. A Mechanical Analysis of Program Verification Strategies. *Journal of Automated Reasoning*, 40(4):245–269, May 2008.

[18] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *Proc. of POPL*, pages 264–276. ACM, 2009.

[19] Z. Yang, K. Hao, K. Cong, S. Ray, and F. Xie. Equivalence checking for compiler transformations in behavioral synthesis. In *Proc. ICCD*, 2013.

[20] Z. Yang, K. Hao, S. Ray, and F. Xie. Handling design and implementation optimizations in equivalence checking for behavioral synthesis. In *Proc. DAC*, pages 117:1–117:6, 2013.

[21] A. Zaks and A. Pnueli. CoVaC: compiler validation by program analysis of the cross-product. In *Proc. of FM*, pages 35–51, 2008.

[22] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. *SIGPLAN Not.*, 47:427–440, Jan. 2012.

[23] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: a methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9:2003, 2003.