Equivalence Checking for Compiler Transformations in Behavioral Synthesis

Zhenkun Yang*, Kecheng Hao*, Kai Cong*, Sandip Ray† and Fei Xie*
* Dept. of Computer Science, Portland State University, Portland, OR 97207, USA
{zhenkun, kecheng, congkai, xie}@cs.pdx.edu
† 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. We present a scalable equivalence checking framework to validate the correctness of compiler transformations employed by behavioral synthesis. Our approach is based on dual-rail symbolic simulation of the input and output design representations of a transformation. We have evaluated our framework on transformations applied to several designs by an open source behavioral synthesis tool, and we present initial results demonstrating the approach.**

I. Introduction

With rapidly increasing complexity in modern VLSI systems, designing high-quality hardware at the Register-Transfer Level (RTL) is challenging. Electronic System Level (ESL) design provides a promising approach to combat this high complexity: design functionality is specified at a high level of abstraction (*e.g.*, with SystemC, C/C++, or domain-specific language), and translated to RTL by a process known as *behavioral synthesis*. Several behavioral synthesis tools are commercially available [1]–[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 description. This task is challenging because of the large difference in abstraction between ESL and RTL.

A behavioral synthesis tool accepts an ESL description of a design and a library of hardware resources, and applies a series of transformations on the former to compile it to RTL. Many of the transformations are generic compiler transformations, geared towards reducing code complexity of the generated design, maximizing data locality, etc. [1]. However, they are aggressively applied to satisfy exacting efficiency needs for synthesized hardware, often under delicate, implicit invariants. Furthermore, the number of transformations is large: it is common that hundreds of compiler transformations are applied during behavioral synthesis. It is unsurprising that the transformations themselves are complex and error-prone, and can lead to buggy synthesized hardware.

In this paper, we present a verification framework to automatically certify compiler transformations in behavioral synthesis through sequential equivalence checking (SEC). Central to our approach is *symbolic simulation*, *viz.*, we symbolically execute the intermediate representation (IR) of the program before and after application of each transformation to certify their semantic equivalence. We demonstrate our framework to certify the compiler transformations applied by the LegUp behavioral synthesis tool [5] on a test suite of 6 designs from various application domains. The most complex design has 188 lines of C, and generates over 15284 lines of RTL.

Compiler certification is an active area of research. Section II surveys some of the literature in the area. Many of these techniques make use of mechanical theorem proving. However, our approach is different because of the uniqueness of our application domain. Furthermore, implementations of the compiler transformations in behavioral synthesis are aggressive, and exploit subtle design invariants to generate highly optimized designs; mechanical certification of such transformations by theorem proving is a non-trivial task. Furthermore, transformation implementations of most commercial synthesis tools are proprietary and not available to mechanical reasoning or to customized instrumentation to generate proof obligations. Thus, we must resort to checking semantic equivalence between the input and output of a transformation without requiring knowledge of the actual implementation. On the other hand, it is well-known that checking equivalence between two arbitrary programs is undecidable. However, for behavioral synthesis, this is ameliorated by restrictions imposed by the synthesis tool. A behavioral synthesis tool accepts designs that can be transformed into a hardware circuit; many generic program features are disallowed, including dynamic memory allocation, system calls, pointer casting, and recursive functions. Finally, our framework can be easily integrated into a commercial synthesis flow since it accounts for (and exploits) the information available from behavioral synthesis tools. In particular, although transformation *implementations* are unavailable, most commercial tools [1], [3] can provide access to the IRs before and after the application of a transformation. We exploit such information to guide SEC without requiring tool vendors to expose implementation details.

The rest of this paper is organized as follows. In Section II, we provide background on behavioral synthesis and symbolic execution, and present related work. In Section III, we propose our framework. We discuss our experimental results in Section IV, and conclude in Section V.

II. Background and Related Work

A. Behavioral Synthesis

A behavioral synthesis tool compiles an ESL description of a hardware design into RTL. Similar to a generic compiler, it first performs lexical, syntax, and semantic analysis, and builds an IR of the ESL description. A series of transformations is

then applied to the IR, which can be categorized into three phases: 1) *compiler transformations*; 2) *scheduling transformations*, which entail computing for each operation the clock cycle for its execution; and 3) *hardware resource binding and control synthesis*. After these transformations, the design can be represented in RTL. The RTL may be subjected to further manual tweaks, optimizing it for different parameters.

B. SEC for Behavioral Synthesis

Previous work [6]–[8] developed a scalable SEC framework for certifying RTL designs generated through behavioral synthesis. The SEC framework compares the generated RTL with the IR obtained after application of compiler and scheduling transformations. However, such a framework is meaningful as a certification mechanism only if the initial compiler and scheduling transformations are correct. Previous work [6] proposed a theorem proving approach to this problem: mechanically verify transformation implementations with a theorem prover, and use these certified implementations for transforming the design representation. However, such an approach is impractical for several reasons. First, the number of transformations is large, *e.g.*, a design can undergo more than a thousand transformations in course of behavioral synthesis. Second, the complexity of their implementations makes theorem proving prohibitively expensive in manual effort. Finally, for most commercial synthesis tools, the transformation implementations are proprietary and consequently unavailable for mechanical reasoning. Our approach addresses these deficiencies through a more automated SEC implementation that is oblivious to implementation details of the transformations.

C. Related Work

There has been research on formally proving compiler transformations correct by theorem prover. One of the first certified compilers was developed as part of the Piton project in the 1980s [9]. More recently, CompCert [10] provides a formally verified compiler of practical complexity. Vellvm [11] project formalizes LLVM's intermediate representation, and develops a framework for reasoning about programs.

Pnueli et al. proposed the notion of *translation validation* [12] for validating the transformations during compilation. Necula used symbolic evaluation techniques from proof-carrying code to tackle translation validation [13].

There has also been recent research on applying symbolic techniques to checking equivalence on software level, *e.g.*, UC-KLEE [14] and SYM-DIFF [15] use symbolic techniques to checking the equivalence of software programs.

III. Equivalence Checking Framework

A. Notations and Definitions

Let P be a program, V be the set of variables of P, and $V_{\mathcal{O}} \subseteq V$ be the set of *observable variables*. Intuitively, variables that we can observe during the execution of P are called observable variables; we assume that the $V_{\mathcal{O}}$ includes the input, output, and global variables.

Definition 1 (State): A state $s \triangleq \{\langle v, u\rangle \mid v \in V,$ u is the value of $v\}$ of a program P is the set of variables in P with their values.

Definition 2 (Observable State): An *observable state* $s_{\mathcal{O}}$ at state s of a program P, denoted by $s_{\mathcal{O}}(s)$, is a projection of s, where variables in $s_{\mathcal{O}}$ are restricted to observable variables in program P.

Remark: We leave the domains for the values of variables undefined for this presentation, but assume that they can be determined from the context. Also, we assume that the domain can be both concrete or symbolic; this permits us to use the same notation for both concrete and symbolic states. For simplicity, we use $s[v]$ to denote the value, either concrete or symbolic, of variable v in state s.

Definition 3 (Path): A *path* $\pi \triangleq s_0, c_1, s_1, c_2, s_2, \ldots, c_n, s_n$ of a program is an alternating sequence of states and state transition conditions, starting from an *initial state* s_0 and ending with a *terminal state* s_n, where c_i is the state transition condition (Boolean expression over program variables) from s_{i-1} to s_i for all $1 \leq i \leq n$.

Definition 4 (Path Condition): Let $\pi \triangleq s_0, c_1, s_1, c_2, s_2, \ldots, s_{n-1}, c_n, s_n$ be a path of a program P. The *path condition* $pc \triangleq \bigwedge_{i=1}^{n} c_i$ of path π is a conjunction of all transition conditions on π. We use $\pi[pc]$ to denote the path condition of π.

Definition 5 (Path Compatibility): Given two programs S and T with the same set of observable variables $V_{\mathcal{O}}$, let π be a path of S with initial state s_0 and path condition pc, and π' be a path with initial state s'_0 and path condition pc' of T. We say π and π' are *compatible* if $s_{\mathcal{O}}(s_0) = s_{\mathcal{O}}(s'_0)$ and $pc \wedge pc'$ is satisfiable. Paths π and π' are called a *compatible path pair* of S and T.

Definition 6 (Path Equivalence): Let π be a path of program S with terminal state s_n, and π' be a path of program T with terminal state s'_m, and suppose that programs S and T have the same set of observable variables $V_{\mathcal{O}}$. We say path π and π' are *equivalent*, denoted by $\pi \sim \pi'$, if π and π' are compatible, and for each variable $v \in V_{\mathcal{O}}$, $s_n[v] = s'_m[v]$.

Informally, two paths are equivalent if they are compatible, and they have the same observable state at their terminal states.

Definition 7 (Program Equivalence): Let S and T be two programs, we say that program S and T are *equivalent*, denoted by $S \sim T$, if every compatible path pair of S and T has the same observable state at their terminal states. Formally let $Paths(S)$ and $Paths(T)$ be all paths of program S and T respectively, program S and T are equivalent if for each path $\pi \in Paths(S)$ and every path $\pi' \in Paths(T)$ that is compatible with π, π is equivalent to π'.

We define the correctness of a transformation by the equivalence of the observable behavior of the source program S and the target program T. Informally S and T are equivalent if feeding the same inputs to both programs, they produce the same output, and have the same effect on the environment (modification to global variables) when terminating.

Definition 8 (Transformation Correctness): Let $\mathcal{T}$ be a transformation which takes a source program S as input and produces a target program T as the output. We say $\mathcal{T}$ is a *correct* transformation on program S if $S \sim T$.

B. Approach Overview

Suppose a transformation $\mathcal{T}$ takes a program S as input and generates a program T as output. We validate the correctness of transformation $\mathcal{T}$ when applied to S by checking that T is equivalent to S. According to Definition 7, we need to prove that S and T has the same observable state at their terminal states for all compatible path pairs.

As a pedagogical simplification, assume that all the paths in S and T are enumerable. Also assume that S and T have the same function signature and global variables; this assumption does not limit our approach because compiler transformations usually do not change function signature unless there are parameters that are irrelevant or unused in the function body, which can be easily detected. Finally, assume that for each observable variable of S we can find the corresponding variable for T and vice versa. We then proceed as follows. We assign the same symbols to the input and non-constant global variables of S and T, then symbolically execute them. After enumerating all paths of S and T, for each compatible path pair π in S and π' in T, we check that π and π' have the same observable behavior; this check is done by an SMT solver by checking the equality between the symbolic expressions of the (symbolic) values of the observable variables.

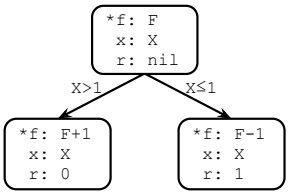
Algorithm 1: CHECK-EQUIVALENCE(S, T)

```
1  s_I ← SYMBOLIZE-INPUTS(S, T)        ▷ Symbolize inputs
2  Π ← SYM-EXE(S, s_I, nil)            ▷ Symbolically Execute S
3  foreach π ∈ Π do
4      s_O ← GET-OBSERVABLE-STATE(π)
5      Π' ← SYM-EXE(T, s_I, π[pc])
6      foreach π' ∈ Π' do
7          s'_O ← GET-OBSERVABLE-STATE(π')
8          if not CMP-STATE(s_O, s'_O, π'[pc]) then
9              print ⟨s_O, s'_O⟩        ▷ Report inequivalences
10             return false
11 return true
```

Algorithm 1 provides a high-level description of our approach. Function CHECK-EQUIVALENCE takes two programs S and T as arguments. Subroutine SYMBOLIZE-INPUTS creates symbols for inputs of S and T. Subroutine SYM-EXE symbolically executes S with symbolic inputs, and collects all paths of S. For each path π with path condition $\pi[pc]$ of S, subroutine GET-OBSERVABLE-STATE collects the observable state s_O corresponding to the terminal state in path π. Subroutine SYM-EXE symbolically executes T with the same symbolic inputs under condition $\pi[pc]$, and collects all paths of T. Since all paths found in T are under the condition $\pi[pc]$, therefore they are all compatible path with π. For each path π' of T found under the condition of $\pi[pc]$, subroutine CMP-STATE checks if π and π' have the same observable state at termination. If the observable states are not equal, the algorithm reports the inequivalences, otherwise it proceeds until all paths of S and T are checked.

Figures 1 and 2 show two programs `foo` and `bar` which are defined in C and their independent symbolic execution trees. Before execution, `foo` and `bar` have the same symbolic input $\{\langle *\mathtt{f}, \mathtt{F}\rangle, \langle \mathtt{x}, \mathtt{X}\rangle\}$, where `F` and `X` are symbolic values of variables `*f` and `x`, respectively. Function `foo` has two paths, the final observable states are $\{\langle *\mathtt{f}, \mathtt{F}+1\rangle, \langle \mathtt{x}, \mathtt{X}\rangle, \langle \mathtt{r}, 0\rangle\}$

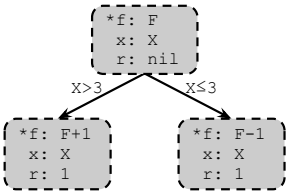
```
1  int foo(int *f, int x){
2    if(x>1) {
3      *f = *f + 1;
4      return 0;
5    } else {
6      *f = *f - 1;
7      return 1;
8    }
9  }
```



(a) (b)

Fig. 1: A simple function `foo` in C with its symbolic execution tree. (a) Function `foo` in C. (b) Symbolic execution tree of `foo`, where `F` and `X` are symbolic values for `*f` and `x`, `r` denotes the return value, and `nil` denotes that the value is not yet available.

```
1  int bar(int *f, int x){
2    if(x>3) {
3      *f = *f + 1;
4      return 1;
5    } else {
6      *f = *f - 1;
7      return 1;
8    }
9  }
```



(a) (b)

Fig. 2: A simple function `bar` in C with its symbolic execution tree. (a) Function `bar` in C. (b) Symbolic execution tree of `bar`, where `F` and `X` are symbolic values for `*f` and `x`, `r` denotes the return value, and `nil` denotes that the value is not yet available.

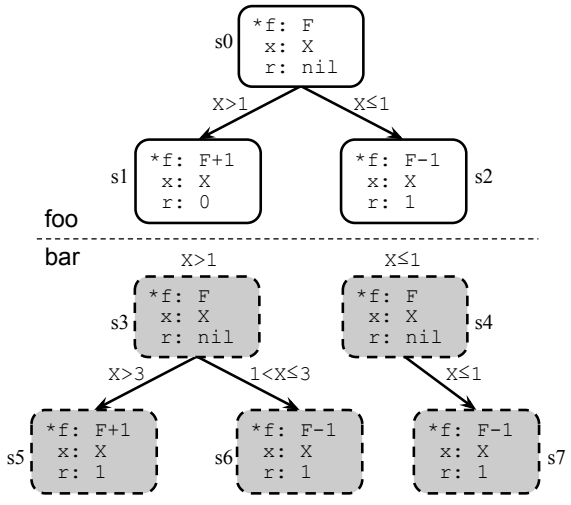


Fig. 3: Symbolic execution tree of function `foo` and `bar`, where `bar` is executed after `foo`, and based on the execution condition of `foo`.

and $\{\langle *\mathtt{f}, \mathtt{F}-1\rangle, \langle \mathtt{x}, \mathtt{X}\rangle, \langle \mathtt{r}, 1\rangle\}$, with conditions $\mathtt{X} > 1$ and $\mathtt{X} \leq 1$, respectively. Similarly, the two final observable states of function `bar` are $\{\langle *\mathtt{f}, \mathtt{F}+1\rangle, \langle \mathtt{x}, \mathtt{X}\rangle, \langle \mathtt{r}, 1\rangle\}$ and $\{\langle *\mathtt{f}, \mathtt{F}-1\rangle, \langle \mathtt{x}, \mathtt{X}\rangle, \langle \mathtt{r}, 1\rangle\}$, with conditions $\mathtt{X} > 3$ and $\mathtt{X} \leq 3$, respectively.

Fig. 3 shows the symbolic execution tree when function `bar` is executed under the path condition of function `foo`. In Fig. 3, state `s0` is the initial state of function `foo`, states `s3` and `s4` are the initial states of function `bar`, states `s1` and `s2` are terminal states of `foo`, and states `s5`, `s6` and `s7` are terminal states of `bar`. We need to conduct three equivalence checks:

- `s1` vs. `s5`, where the return values are not equivalent;
- `s1` vs. `s6`, where the values of $*\mathtt{f}$ are not equivalent, and return values are also not equivalent;
- `s2` vs. `s7`, the states are equivalent.

Therefore, our checking algorithm returns that `foo` and `bar` are not equivalent, and reports the inequivalences.

IV. Experimental Results

As initial demonstration of the viability of the approach, we have applied it for validation of transformations on a test suite of 6 designs. All designs are implemented in C. We used the open-source LegUp behavioral synthesis tool [5] to synthesize the designs. Table I shows the complexity of these designs.

TABLE I: Summary of Experimental Results

App. Domain	Cryptography		Signal Processing		Image Processing	
Design	TEA	SHA-1	FIR	FFT	YUVToRGB	DCT
Lines of C	11	33	37	188	20	48
Lines of RTL	1010	5016	1145	15284	841	1708
# Transformations	10	18	14	22	11	6
# Successful Checks	8	16	14	19	7	6
Success Rate (%)	80	88.89	100	86.36	64.64	100
Avg. Time (s)	1.3	4.6	1.1	10.9	1.2	1.7
Memory (MB)	9.87	18.26	9.59	11.54	9.90	9.93

We conducted our experiments on a laptop with Debian 6.0.6 running on a 2.66 GHz Intel dual-core i7 processor with 8 GB of memory. Our experiments were run with a cutoff time of 60 seconds; certifications that take longer than this time are classified as failures. The reason is that most successful transformation certifications were empirically found to complete within 10 seconds, if at all; if symbolic execution takes more than 60 seconds, it is unlikely to complete in any reasonable time. Based on this observation, our projection is that the impact of making the cutoff longer on the number of successful transformations would be insignificant.

Table I shows the statistics of our experiments, *viz.*, the number of transformations that the behavioral synthesis tool applied,[1] the number of transformations we can successfully check, the rate of successful checks, and the time and memory usages. In all successful cases, the running time and memory usage are moderate. We successfully validated 86 percent (70 out of 81) of transformations in total. We can check all transformations on `FIR` and `DCT` designs. We were not able to check 11 out of 81 transformations. This is mainly because every design in our test suite has loop structures, except `DCT`; if the transformation changes some operations inside the loop, the expression sent to the SMT solver may be very long since it will correspond to the computation of the unrolled loop. Therefore the solver fails to return the result within the time limit (60 seconds in our experiments).

[1] Some transformations are applied more than once by the synthesis tool.

V. Conclusion and Future Work

We have presented an equivalence checking framework to validate the correctness of compiler transformations in behavioral synthesis. We use symbolic execution technique to explore (possibly all) paths of the source and target program of each transformation. Although simple, our framework shows promise, *e.g.*, in our initial experiments we can automatically certify more than 86% of transformations applied by a behavioral synthesis tool on our test suite.

In future work, we plan to extend our SEC framework to handle more aggressive transformations. Our planned future extensions also include handling unbounded loop structures and designs with a large number of function calls.

VI. 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 their advice and help.

References

[1] 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*, vol. 30, no. 4, pp. 473–491, April 2011.

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

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

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

[5] 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*, 2011, pp. 33–36.

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

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

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

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

[10] X. Leroy, "A formally verified compiler back-end," *J. Autom. Reason.*, vol. 43, no. 4, pp. 363–446, Dec. 2009.

[11] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formalizing the LLVM intermediate representation for verified program transformations," *SIGPLAN Not.*, vol. 47, no. 1, pp. 427–440, Jan. 2012.

[12] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *Proc. of TACAS*, 1998, pp. 151–166.

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

[14] D. A. Ramos and D. R. Engler, "Practical, low-effort equivalence verification of real code," in *Proc. of CAV*, 2011, pp. 669–685.

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