Microprocessors and Microsystems 37 (2013) 1108–1121

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Formal equivalence verification and debugging techniques with auto-correction mechanism for RTL designs

Bijan Alizadeh *, Payman Behnam

School of Electrical and Computer Engineering, College of Engineering, University of Tehran, Tehran 14395-515, Iran

ARTICLE INFO

Article history:
Available online 19 October 2013

Keywords:
Equivalence checking
Formal verification
Debugging
RTL designs

ABSTRACT

By increasing the complexity of system on chip (SoC) designs formal equivalence verification and debugging have become more and more important. Lower level methods such as BDDs and SAT solvers suffer from space and time explosion problems to match sizes of industrial designs in formal equivalence verification and debugging. This paper proposes techniques to verify and debug datapath intensive designs based on a canonical decision diagram called Horner Expansion Diagram (*HED*). It allows us to check the equivalence between two models in different levels of abstraction, e.g., a Register Transfer Level (RTL) implementation and a non-cycle-accurate specification. In order to reduce the complexity of equivalence checking problem, we tackle the exponential path enumeration problem by automatically identifying internal equivalent conditional expressions as well as suitable merge points. Our debugging technique is based on introducing mutations into the buggy implementation and then observing if the specification is capable of detecting these changes. We make use of a simple heuristic to reduce the number of mutants when dealing with multiple errors. We report the results of deploying our equivalence verification technique on several industrial designs which show 16.8× average memory usage reduction and 8.0× speedup due to merge-point detection. Furthermore, our debugging technique shows 13.7× average memory usage reduction and 4.6× speedup due to using SMT solvers to find equivalent conditions. In addition, the proposed debugging technique can avoid the computation of unnecessary mutants so that the results show 2.9× average reduction of the number of mutants to be processed.

1. Introduction

With the increased size and complexity of digital systems design verification and debugging become a dominating factor of the overall digital design flow. Sequential Equivalence Checking (SEC) is a process of formally proving functional equivalence of designs that may in general have sequentially different implementations. Examples of sequential differences span the space from retimed pipelines, differing latencies and throughputs, and even scheduling and resource allocation differences. On the other hand, debugging is a process of finding errors or bugs in the RTL implementation so that the bugs can be removed to make the RTL design function in the way it was desired. In order to match sizes of real world designs in formal equivalence checking and debugging, reducing run times and memory usage for computations is a key point. Most of hardware verification tools however are based on bit-level methods like BDD or SAT solvers that suffer from space and time explosion problems when dealing with industrial designs.

Many companies have paid more attention to design hardware at higher levels of abstraction due to faster design changes and higher simulation speed. In this phase, a C-like high level specification is described and then refined to a RTL design by adding more and more implementation details at different steps. Therefore, there is a significant increase in the amount of verification required to achieve functionally correct description at each step, if traditional dynamic techniques such as simulation are used. This has led to a trend away from dynamic approaches and therefore SEC methods have become very important to reduce time-to-market as much as possible.

A few non-cycle-accurate sequential equivalence checking approaches have been proposed. In symbolic simulation based approaches [1–6], loop and conditional statements need to be unrolled and then all paths through the code must be explored. The hope is that the individual cases are much easier to solve than the original problem. However, if dependencies exist between different iterations of a loop statement, it will increase the run time for symbolic simulation and degrades quality due to the exponential number of paths. As an example, consider equivalence checking of the two C-code snippets in Fig. 1(a). After unrolling *for-loop*, we encounter the exponential path enumeration problem because corresponding to each *then* and *else* branch it is necessary to

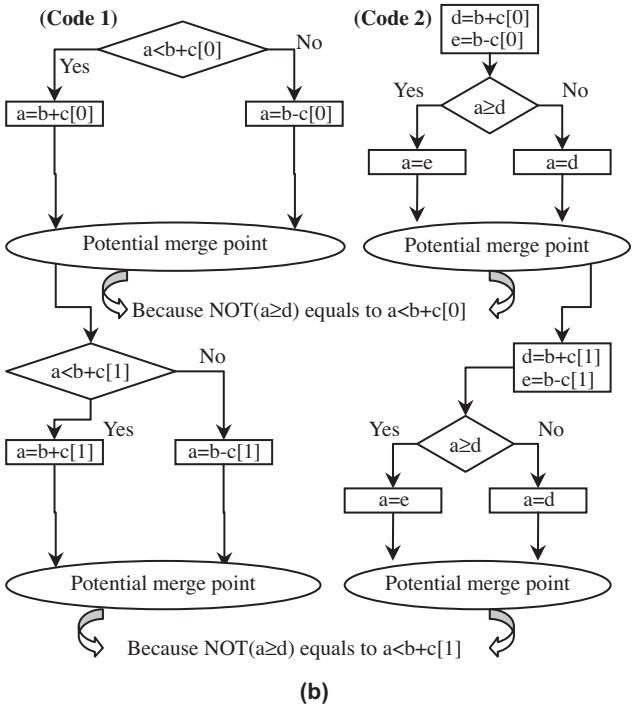
* Corresponding author.
E-mail addresses: b.alizadeh@ut.ac.ir (B. Alizadeh), payman.behnam@ut.ac.ir (P. Behnam).

0141-9331/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.micpro.2013.10.002

```
for (i=0; i<N; i++)                    (code 1)
    if ( a < b + c[i])    a = b + c[i];
    else                  a = b - c[i];
```

```
for (i=0; i<N; i++)                    (code 2)
      d = b + c[i];     e = b - c[i];
  if (a >= d)    a = e;    else     a = d;
```

(a)



(b)

Fig. 1. Path enumeration of conditional statements (a) original source codes and (b) potential merge point if the equivalence of conditional expressions can be checked.

have two execution paths which results in enumerating 2^N paths where N is the number of iterations. It is also to be noted that the results computed on the different paths must be tracked, which will cause a blow-up in logic if lower level techniques such as BDDs and SAT solvers are utilized.

The basic idea to cope with this complexity is the fact that many problems become significantly easier by case-splitting on the right input variables or internal conditions. In this paper, we present a novel technique to automatically find equivalent internal conditions, e.g. $a < b + c[i]$ and NOT($a \geqslant d$) in Fig. 1(b), so that the equivalence verification problem of cycle-accurate implementation and un-timed specification is reduced into much easier individual equivalence checking problems which are easily solved. Please keep in mind that both *code* 1 and *code* 2 in Fig. 1 are un-timed models (algorithmic level descriptions) in order to clarify our merge-point detection mechanism, while our verification and debugging techniques can deal with cycle-accurate implementation and un-timed specification as will be discussed in Sections 4 and 5.

In summary, the main contributions of this paper compared to our previous works [7–11] are as follows:

- Automatic finding of internal equivalent conditional expressions as well as suitable merge points by using SMT solvers and our decision diagram called Horner Expansion Diagram (*HED*) to overcome exponential path enumeration problem while the process of finding equivalent conditional expressions in [8] is done manually. We make use of the *HED* graph as a canonical representation to check the equivalence between computationally expensive designs at different levels of abstraction [12,13].
- Adapting software mutation-based debugging technique into the RTL designs. In the case of nonequivalent behavior of the RTL and algorithmic level descriptions, our debugging technique helps to locate and correct the bug quickly due to converting the debugging problem to an equivalence checking problem by introducing mutations. Unlike our previous debugging technique presented in [11], in this work a simple heuristic is presented to reduce the number of mutants when multiple errors need to be considered.
- Showing empirical results to prove that these techniques allow us to provide appropriate equivalence verifications and highly accurate diagnoses very quickly.

The rest of this paper is organized as follows. In Section 2, we address related work to highlight the importance of the RTL formal verification and debugging. To make our paper self-contained, the *HED* [12,13] as a hybrid canonical representation is described in Section 3. The proposed *HED*-based formal equivalence verification and debugging techniques are presented in Sections 4 and 5, respectively. Finally, experimental results and a brief conclusion and future work are shown in Sections 6 and 7, respectively.

2. Related work

2.1. Equivalence checking techniques

Recently, some techniques have been proposed to apply equivalence checking to the system level and the RTL descriptions [1–6]. In [1] an equivalence checking technique to verify system level design descriptions against their implementations in RTL was proposed. It presented an automatic technique to compute high level sequential compare points to compare variables of interest in the candidate design descriptions. They start the two design state machines at the same initial state and step the machines through every cycle, until a sequential compare point is reached. At this point the equivalence of the two state machines is proved using a lower (Boolean) level engine which is zChaff Satisfiability (SAT) solver. One of the limitations of this technique is not to be scalable in the number of cycles. As the number of cycles gets larger, the size of the expression grows quadratically, causing capacity problems for the lower level SAT engine. Furthermore it may not be applicable to large designs due to arithmetic encoding. In addition, in this technique corresponding equivalent points between two descriptions should be determined while these points may not be at all obvious due to complex control flow.

The authors of [2] have proposed early cut-point insertion for checking the equivalence of high level software against RTL of combinational components. They introduce cut-points early during the analysis of the software model, rather than after generating a low level hardware equivalent. This way, they overcome the exponential enumeration of software paths as well as the logic blow-up of tracking merged paths. However, it is necessary to synthesize word level information into bit level because of using BDD to represent the symbolic expressions and so the capacity is limited by memory and run time requirements. In addition, it has only focused on combinational equivalence checking and has not addressed how to extend the proposed method for sequential equivalence checking problem. Another approach to equivalence checking between C descriptions is presented in [3]. This approach