Behavior-RTL equivalence checking based on data transfer analysis with virtual controllers and datapaths

Masahiro Fujita

VLSI Design and Education Center, The University of Tokyo
2-11-16 Yayoi, Bunkyo-ku, Tokyo, 113-0032, Japan
fujita@ee.t.u-tokyo.ac.jp

Abstract. A behavior-RTL equivalence checking method based on bottom-up reasoning is presented. Behavior and RTL descriptions are converted into dependence graphs from which virtual controllers/datapaths are generated. Actual equivalence checking is based on isomorphism analysis on dependence graphs and also virtual controllers/datapaths. First equivalence classes on partial computations are extracted by using Boolean reasoning on virtual controllers/datapaths. Then these equivalence classes are used to prove the equivalence of the entire descriptions in a bottom-up way.

1 The proposed verification method

In this paper, we propose a way to verify equivalence by establishing mappings between behavior and RTL descriptions. We first extract "classes of equivalent partial computations". Using these accumulated correspondences, the equivalence checking problem can be solved by establishing mappings on the entire design descriptions followed by reasoning about them in a bottom-up fashion. This is a similar technique to combinational equivalence checking methods based on internal equivalent points, such as the one in [1]. We map given behavior and RTL descriptions into virtual controllers and datapaths [2] and then reason about those design descriptions. The virtual controllers and datapths can make it possible to separately reason about "timing" and "data computations" and can establish correspondence among partial computations in a bottom-up way. Our verification methods have four steps as follows:

(Step 1) Generate system dependence graph (SDG), which represents dependencies among statements in design descriptions, and virtual controllers/datapaths from both behavior and RTL descriptions

(Step 2) Gather information on equivalence classes on partial computations on SDG. In this step, if necessary equivalence classes are computed by analyzing virtual controllers/datapaths as well as SDG. When analyzing virtual controllers/datapaths, apply reachability computation on virtual controllers to decide equivalence of partial computations.

(Step 3) Perform graph matching between the SDGs for behavior and RTL descriptions by using equivalence classes computed in (Step 2).

(Step 4) If the result of (Step 3) gives matching on SDGs, we conclude that the behavior and RTL descriptions are equivalent. Otherwise go back to (Step 2),

and try to get more equivalence classes. If no more equivalence classes are available, we generate a computation path which differentiates computations in the two SDGs as a counter example.

Please note that the counter example generated in (Step 4) may not be a real computer example, since in (Step 2) we may not be able to gather all equivalence classes. That is, there are cases where our results are false-negative.

The system dependence graph that we are using in the proposed equivalence checking method is generated by program slicers. Program slicing [3] is a technique by which related portions of the programs are extracted based on user-specified criteria. In the program slicing tools, internally control flow graphs and also so called system dependence graphs (SDG) are generated. SDG represents all static dependencies among statements in terms of control, data, and interference. In our method, we are using control flow graphs and SDGs generated by program slicers when generating virtual controllers and virtual datapaths. Our program slicer [4] is targeting SpecC language [5] and also C/C++ descriptions, and so combined descriptions in those languages can also be processed. The slicing program generates the corresponding control flow graphs (CFGs) and system dependence graphs (SDGs) as a unified graph. Then they are further processed to generate virtual controllers/datapaths.

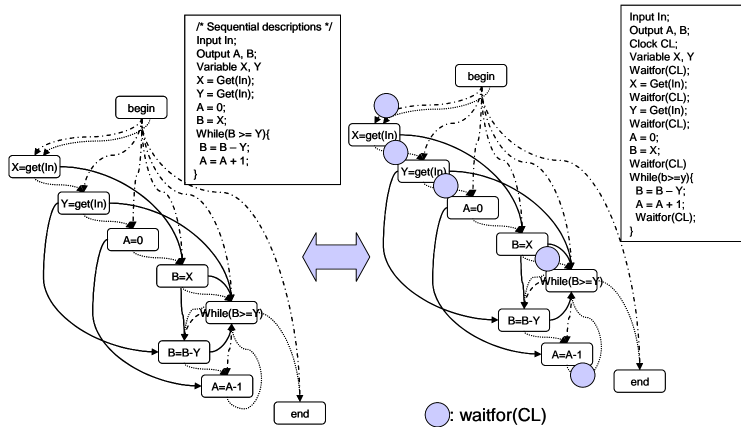


Figure 1. Behavior and RTL descriptions and their System Dependence Graph (SDG)

Now we illustrate the above verification method with examples. An example behavior description and a corresponding RTL one that is supposed to implement the behavior description are shown in the boxes of Figure 1. They are computing divisions, and the first input from the input port, In, is divided by the second input form the input port, and the output, A, is the quotient and the output B is the remainder at the end of the computation. The division is very straightforwardly computed by counting up how many times the value of the divider can be extracted. The semantics of the descriptions are obvious from the descriptions, and we do not explain them here except for "waitfor(CL)" statement. It is the statement that determines the clock boundary in RTL descriptions to fix the scheduling of the RTL descriptions. All statements surrounded by neighboring two waitfor(CL) statements must be executed within the same clock cycle. Since waitfor(CL) statements are the only difference, these two descriptions should be recognized to be equivalent. However, the values of

output signals, A and B, may not be equal for every clock cycle, since behavior description has no fixed scheduling in the terms of clock timing. So in this case we assume that with an appropriate use of attribute statements [2] what should be compared is defined as to check the values of the outputs at the end of computation only. The SDGs for the two descriptions are also shown in Figure 1. The difference is just the existence of several "waitfor(CL)" statements in RTL, and so the two SDGs are easily recognized as "matching", that is, they are isomorphic other than nodes for "waitfor(CL)". We basically use graph isomorphism check for identifying equivalence of computations, and equivalence classes are used to make matching on sub-graphs..

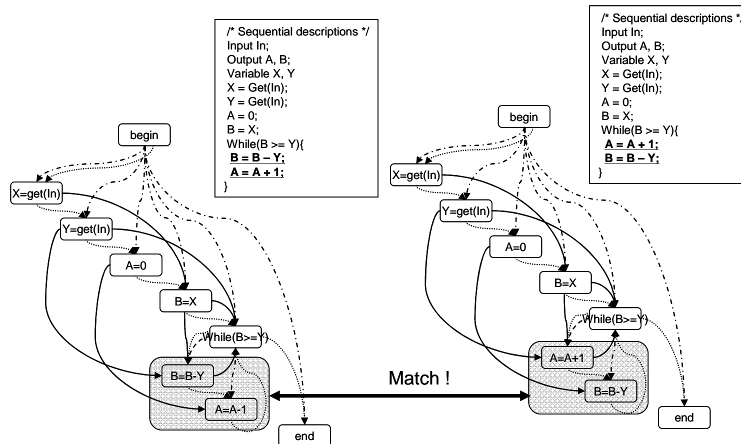


Figure 2. Identification of equivalences of subgraphs in SDG in a bottom-up way

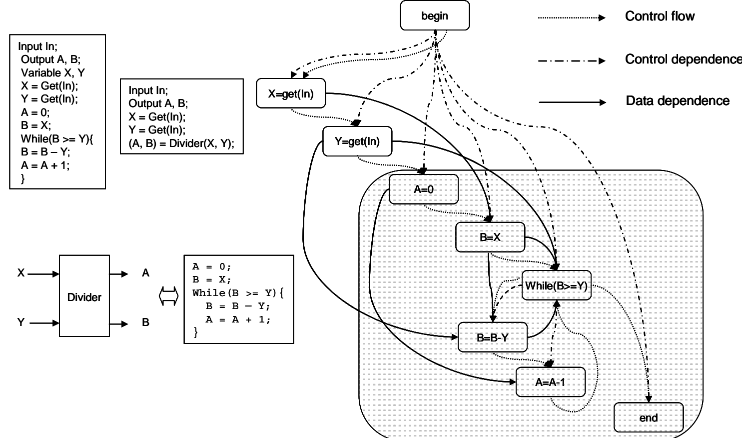
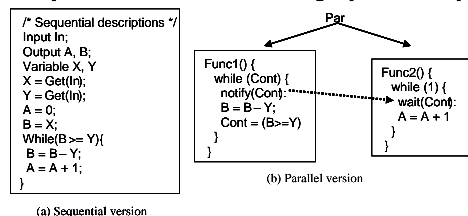


Figure 3. Bottom-up reasoning by identifying sub-graphs for "division" circuit/computation

For more complicated cases, first of all equivalence class are first computed as explained in (Step 2) in the previous section. For example, the equivalence checking on the two descriptions shown in Figure 2 is processed as follows. Here we are comparing the two descriptions inside the boxes. The only difference between the two is the order of executions of the two underlined statements. Since they are independent with each other, these statements compute exactly the same. This can be

easily checked by traversing the SDGs and make sure they are independent. Then we can have an equivalence class for these statements and use it for the comparison of the two SDGs generated from the descriptions as shown in Figure 2. After identifying the equivalence class, the two SDGs are isomorphic and so the descriptions are equivalent. Figure 3 shows a more complicated case. In this example, the portion of the original description for division computation is replaced by a divider circuit as shown in the left-top part of the figure. First of all, we try to prove with loop-invariants that the while-loop part in the original description is computing division. With appropriate loop invariants, we can decompose the verification problem for the while-loop into the ones for non-loops. The decomposed verification can be processed as Boolean reasoning problems with virtual controllers/datapaths. Once that is finished, the equivalence for the entire SDGs can be again by checking their isomorphism.
Figure 4 shows another example between sequential and parallel descriptions. In such cases, we first extract sequential behaviors from parallel ones by identifying synchronization statements, such as "notify" and "wait" and using them to generate sequential orders of executions. The extracted behaviors are then compared with the original sequential ones in terms of graph isomorphism utilizing equivalence classes.



(a) Sequential version

(b) Parallel version

Figure 4. **Sequential and parallel description comparison by first extracting sequential behaviors from parallel ones**

2 Experimental results

We have tried several SpecC descriptions, such as the SpecC examples shown in SpecC manuals, e.g., elevator system, parity checker, and so on. Also, we have verified two versions of internet PPP protocol descriptions. These examples are ranging from one hundred to a couple of thousands lines of SpecC codes. Also, in the case of designs generated by SoC Environment, a system level design/synthesis tool developed by UC Irvine [5], the difference between two successive synthesis steps in the tool is very limited, and the analysis on partial computation equivalence classes becomes very simple but very useful. Several tens of thousands lines of SpecC descriptions can be verified with the proposed methods for such cases, including description on MPEG4 encoders.

References

[1] A. Kuehlmann and F. Krohm, "Equivalence Checking Using Cuts and Heaps", in Proceedings of the 34th ACM/IEEE Design Automation Conference 1997.
[2] M. Fujita, "On equivalence checking between behavioral and RTL descriptions", HLDVT 2004, Nov. 2004. Also see http://www.cad.t.u-tokyo.ac.jp for more reference.
[3] M. Weiser, "Program slices: Formal, psychological, and practical investigations of an automatic program abstraction", PhD thesis, University of Michigan, 1979.
[4] K. Tanabe, S. Sasaki, M. Fujita, "Program slicing for system level designs in SpecC", IASTED Conference on Advances in Computer Science and Technology, Nov. 2004.
[5] SoC Environment, University of California, Irvine. http://www.cecs.uci.edu/~cad/sce.html.