Matching Abstract and Concrete Hardware Models for Design Understanding

Tino Flenker*
*Institute of Computer Science, University of Bremen
28359 Bremen, Germany
Email: flenker@informatik.uni-bremen.de

Görschwin Fey*†
†Institute of Space Systems, German Aerospace Center
28359 Bremen, Germany
Email: goerschwin.fey@dlr.de

***Abstract*—Nowadays, before a microchip's concrete implementation is available a more abstract model, e.g., on *electronic system level* (ESL) is created. To ensure a better design understanding a matching of both model's variables is proposed. But how to map a variable from the abstract model to a variable form the concrete model? We evaluate a simulation based approach to address this problem. We instrument both models to get traces for each variable and propose three methods to figure out which variable matches to a corresponding variable of the other model.**

I. Introduction

Microchip's complexity nowadays increases at tremendous speed. To adhere to strict time-to-market constraints tools are required which facilitate a rapid understanding and incorporation of hardware designs. That makes *design understanding* (DU) an important research topic, because DU enables a faster debugging by tool support. A faster understanding supports new colleagues in a company and the training period can be reduced.

To solve *equivalence checking* (EC) several approaches are proposed. In [1][2] formal methods are used but it is not feasible to perform EC using conventional equivalence checkers due to significant internal differences in abstract and concrete model.

Our approach uses a simulation based approach for EC. Using simulation to find potentially equivalent nodes in two circuits is common as a preprocessing step for formal EC. Such equivalent nodes are often called cut-points. This also holds when comparing abstract and concrete models [3][4][5][6]. For EC complete equivalence of the models is expected often including cycle accuracy [7]. However, for design understanding the two models are expected to be quite different. Usually in an abstract model the timing information is lost and cannot be used. Because of that, traces will have a different length and simple matching by comparing values one by one is not possible.

In [8] the authors propose a method for non cycle accurate EC. However, this approach is only suitable for *register transfer level* (RTL) to RTL EC so it is not relevant for ESL to RTL EC. The authors of [9] use a simulation-based approach but assume that both models are available in SystemC [10]. Our approach aims to find corresponding parts of the abstract model in the concrete model available in Verilog.

For DU we propose an approach to match abstract models with concrete models. By this, the designer can directly find the implementation of abstract functionality. Matching models is reduced to mapping variables between the two abstraction levels. We use simulation traces to perform the mapping. The underlying assumption is that variables relating to the same functionality yield similar simulation traces. In the following we present three methods, which compare traces of concrete and abstract models. The methods handle traces despite of absence of timing information and despite of different lengths. Experimental results show the quality of the approach.

II. Methodology

This section describes the proposed approach. First, the work flow is presented and afterwards three methods for variable matching are explained.

A. Work Flow

The main goal is to find relations between variables of an abstract and a concrete hardware model. For that an implementation written in a *hardware description language* (HDL) and another implementation written in a higher level programming language like C/C++ or SystemC are considered. An example for an abstract implementation is an *instruction set simulator* (ISS) of a processor.

To get a trace of each implementation some use cases are needed, which execute the same functionality and are available for both models. To get the traces, the hardware model can be simulated and the values of registers and internal signals can be printed. To get the abstract model's trace, the model can be instrumented so that the values can be printed, when needed.

For example, consider a processor and a corresponding ISS. In addition a compiler and a program in C/C++ are needed. Next, the program is compiled and on each implementation the program is executed and a trace from each model is generated.

B. Trace Analysis

In this section three methods for trace analysis are introduced. In short they are described as follows:

1) Get the set of each variable's values.
2) Get the set and count the number of occurrences of each variable's values.
3) Consider the sequence of variable's values.

The methods can also be used one after another. That means, all indistinguishable variables after the first method is finished are the input for the second method. Next, all remaining

This work was supported by the University of Bremen's Graduate School SyDe, funded by the German Excellence Initiative, and the German Research Foundation (DFG, grant no. FE 797/6-2).

This work has also been partially funded from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644905 (IMMORTAL).

t	DataIn		
1	0xff		
2	0xff		0xff: 2
3	0xfc		0xfc: 1
4	0xff	$\rightarrow$	0xff: 3
5	0xff		0xac: 1
6	0xff		
7	0xac		

Figure 1. Folding a trace

variables after the second analysis constitute the input for the last method.

1) Set: The first method collects all values of each variable in a separate set. To match traces, the symmetric difference is taken. The set from the variable with the smallest symmetric difference is the best fit which is considered as a candidate. If the best fit is computed for multiple variables, then each of these variables are the candidates.

Here, it is assumed that signals for data get the same values in both implementations. Consequently the result is the same set of values. Hard to distinguish are control signals which typically consist of a bit width of one. The sets of these signals contain at most the values 0 and 1. In this way, a distinction of the control signals is impossible.

2) Count: The second method encounters the drawback of method one. This method assumes that different control signals are toggled a different number of times. In this manner the control signals can be distinguished. This method collects all values of the different variables, too. But in addition to that, each occurrence of a value is counted for each variable.

To match traces by this method, first the symmetric difference is also calculated. Traces with big differences are no longer considered for this method. For further steps, the intersections of the sets are examined. In addition a factor is calculated to align traces of different lengths. The factor is calculated by the number of operations of the first trace divided by the number of operations of the second trace. Next, for each value the difference after the alignment is computed. For alignment the value's number of occurrences is normalized using the aforesaid factor. Then, for normalized occurrences of each value the difference of the first trace and the aligned trace is calculated. The sum of all differences is called divergence. The variables with the trace of smallest divergence are the candidates for the match.

3) Sequence: The third method additionally considers the sequence of values. Here, the values for each variable are *folded* and stored in order of occurrence. This is shown in Fig. 1. Folding means if the value of a variable was not changed over several clock ticks all equal values will be collected and the number of folded lines is stored. One idea in our heuristic is to match variables by considering sequences of values. Two sequences with equal values of data indicate two variables representing the same semantics on the functional level. Moreover, if a signal stays for a long time on one value, the corresponding variable in the other design needs to stay on the same value for a long time, too.

Here the matching is computed as follows. Both traces are observed value by value. If two values are not equal, then the next identical value is determined. If no identical value exists, the traces becoming discarded. For all remaining

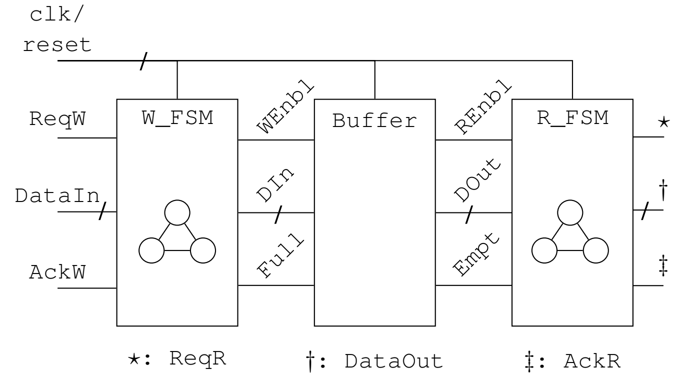

Figure 2. Bus bridge implementation for experiments

traces a penalty value is calculated. First, the penalty includes the difference between length of folded traces. Next, the differences of normalized occurrences of the first and the aligned trace is added to the penalty for each value.

III. RESULTS

This section summarizes the experimental results. For experiments a small bus bridge model is implemented (see Fig. 2). On the inputs a write (ReqW) or read (ReqR) request can be made. In the same clock tick the data is read (DataOut) from or written (DataIn) to the buffer. One clock tick later the model acknowledges whether data was read (AckR) from or written (AckW) to the buffer. Two *finite state machines* (FSMs) control the internal buffer and protect against writing into a full buffer (W_FSM) and reading from an empty buffer (R_FSM).

The abstract model is implemented in C++. The model is a *class* and the buffer is represented by a queue of the standard library. The concrete model is implemented in Verilog. The model is *module* and the buffer is represented by memory address pointer.

Table I. MATCHING OF VARIABLES BY METHODS

	set	count	sequence
ReqR	ReqR,ReqW,AckR,AckW	ReqR	ReqR
ReqW	ReqR,ReqW,AckR,AckW	ReqW,AckW	ReqW,AckW
DataIn	DataIn	DataIn	DataIn
AckR	ReqR,ReqW,AckR,AckW	AckR	ReqR
AckW	ReqR,ReqW,AckR,AckW	ReqW,AckW	ReqW,AckW
DataOut	DataOut	DataOut	DataOut

Use cases which write to the buffer and read the results are used. Table I presents the results of the experiments. The headline shows which method is represented by the given column. The first column lists the considered variables. The trace of the given variable is compared to each trace of the other model and the content of each cell lists the variables with the best matching by the metrics shown in Section II-B.

Table I shows a good matching is achieved for the data in- and outputs with each method. The results for the control signals are not so clear. Taking only the set of values results in a list of all control signals for all control signals. That is obvious because the sets only consist out of 0 and 1 and sets consisting of the same values are not distinguishable. To count the number of occurrences for each value leads to better results. The control signals for the read operation (ReqR, AckR) are uniquely identified. For a distinction of the signals for write operations the method seems not to be mature enough. Both variables (ReqW, AckW) respectively match each other. An

extension of the use cases could cause an improvement. The third method which considers the sequence of the values leads to the same results for the write operation. However, both variables of the read operation match to the same variable (`ReqR`) which shows a deterioration to the second method, because the matching of `AckR` by the counting method is correct and here it is not.

REFERENCES

[1] K. Hao, F. Xie, S. Ray, and J. Yang, "Optimizing equivalence checking for behavioral synthesis," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, March 2010, pp. 1500–1505.

[2] S. Vasudevan, V. Viswanath, J. A. Abraham, and J. Tu, "Sequential equivalence checking between system level and rtl descriptions," *Design Automation for Embedded Systems*, vol. 12, no. 4, pp. 377–396, 2008.

[3] A. Finder, J. Witte, and G. Fey, "Debugging HDL designs based on functional equivalences with high-level specifications," in *IEEE International Symposium on Design and Diagnostics of Electronic Circuits Systems*, 2013, pp. 60–65.

[4] B. Alizadeh and M. Fujita, "Automatic merge-point detection for sequential equivalence checking of system-level and rtl descriptions," in *Proceedings of the International Conference on Automated Technology for Verification and Analysis*, 2007, pp. 129–144.

[5] X. Feng and A. J. Hu, "Early cutpoint insertion for high-level software vs. RTL formal combinational equivalence verification," in *Proceedings of Design Automation Conference*, 2006, pp. 1063–1068.

[6] C. Karfa, C. Mandal, D. Sarkar, S. R. Pentakota, and C. Reade, "A formal verification method of scheduling in high-level synthesis," in *Proceedings of the International Symposium on Quality Electronic Design*, 2006, pp. 71–78.

[7] S. Vasudevan, J. Abraham, V. Viswanath, and J. Tu, "Automatic decomposition for sequential equivalence checking of system level and rtl descriptions," in *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, 2006, pp. 71–80.

[8] P. Chauhan, D. Goyal, G. Hasteer, A. Mathur, and N. Sharma, "Non-cycle-accurate sequential equivalence checking," in *Design Automation Conference*, ser. DAC '09. New York, NY, USA: ACM, 2009, pp. 460–465.

[9] D. Große, M. Groß, U. Kühne, and R. Drechsler, "Simulation-based equivalence checking between systemc models at different levels of abstraction," in *Great Lakes Symposium on VLSI*, 2011, pp. 223–228.

[10] "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, Jan 2012.