Synchronized Debugging across Different Abstraction Levels in System Design*

Rolf Drechsler[1,2], Daniel Große[1], Hoang M. Le[1], and André Sülflow[1]

[1] Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2] Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
drechsle@informatik.uni-bremen.de

Abstract. The development of complex systems such as today's System-on-Chips (SoCs) under tight time-to-market constraints is an extremely challenging task. To cope with the rapidly increasing complexity, the level of abstraction has been raised beyond RTL to the so-called Electronic System Level (ESL). In typical ESL design flows, the design is started from the textual specification. Then the golden model is built using abstract ESL languages such as SystemC. This model is used by several teams focusing on different aspects, like performance analysis and early software development. At the same time the refinement process starts and IP components are integrated. Finally, an RTL model is built for the hardware part of the system. However, in practice the design process is not an ideal incremental top-down design process. Therefore, bugs can be found during the different refinement steps. Hence, it is important to have the respective models in sync. In this paper we consider debugging approaches at different levels of abstractions and show how to relate the debugging results across the abstraction levels. Following this methodology, the productivity of the whole design process is accelerated because each team always works on the latest correct design.

1 Introduction

Although embedded systems have witnessed a reduction of their development time and life time in the past decades, their complexity has been increasing steadily. As a result, the development of embedded systems moves from design to verification, i.e. more time is spent on checking whether the developed design is correct or not. In fact, according to a recent study [1], from 2007 to 2010, there has been a 4% increase of designers compared to an alarming 58% increase of verification engineers.

To face the respective verification challenges, significant effort has been put into clever verification methodologies and new flows have been investigated. A major milestone for the development and verification of embedded systems has

* This work was supported in part by the Federal Ministry of Economics and Technology (BMWi), Germany, within the EXIST Transfer of Research project SolVerTec. For more details see www.solvertec.de.

become the so-called *Electronic System Level* (ESL) design which is state-of-the-art today [2]. Here, the idea is to start designing a complex system at a high level of abstraction – typically using an algorithm specification of the design. At this level, the functionality of the system is realized and evaluated in an abstract fashion ignoring e.g. which parts might become hardware or software later.

The next level of abstraction is based on *Transaction Level Modeling* (TLM) [3]. As modeling language typically SystemC [4,5,6] is used which offers the TLM-2.0 standard [7]. A TLM model consists of modules communicating over channels, i.e. data is transferred in terms of transactions. Within TLM, different levels of timing accuracy are available such as untimed, loosely-timed, approximately-timed, and cycle-accurate. The respective levels allow e.g. for early software development, performance evaluation, as well as HW/SW partitioning and, thus, enable a further refinement of the system.

Finally, the hardware part of the TLM model is refined to the *Register Transfer Level* (RTL), i.e. a description based on precise hardware building blocks which can subsequently be mapped to the physical level. Here, the resulting chip is eventually prepared for manufacturing.

However, in practice the design process is not an ideal incremental top-down design process. Models at different abstraction levels are being used in parallel by several teams focusing on different aspects. Therefore, in the design process, bugs can be found during the different refinement steps. Furthermore, the same bug can exist simultaneously at several abstraction levels due to refinements of a buggy model. Hence, it is important to have the respective models in sync and to minimize the debugging effort.

In this work, we introduce the concept of *synchronized debugging*. Once a bug has been detected at an abstraction level, the subsequent debugging process tries to find and fix the bug not only in the respective model, but also at the other abstraction levels. We assume that for a detected bug, a counter-example (or a failing testcase) is available. Synchronized debugging also requires that during the refinement steps, the correspondence between the original and the refined elements is kept. The process starts with applying specialized debugging approaches for the abstraction level where the bug has been found. Once the bug is localized and fixed in the respective model, it is continued with other abstraction levels. First, the counter-example is translated for each of these abstraction levels. Then, the validity of the refined (or abstracted) counter-example is checked to confirm whether the same bug also exists at that abstraction level. If so, the debugging result at the abstraction level where the bug has been found can be reused. Based on the established correspondence, the bug can be localized and fixed for the considered abstraction level without using specialized approaches. Thus, following the outlined methodology, models at different levels of abstraction are kept in sync and multiple separated debugging processes are not necessary.

In the next section, we demonstrate the ideas by means of an example at two representative abstraction levels.

```
1  void calculate(calc_payload& p) {
2      p.calc_status = CALC_OKAY;
3      switch (p.op) {
4          case NOP : break;
5          case ADD : acc = p.number1 + p.number2; break;
6          case SUB : acc = p.number1 − p.number2; break;
7          case MULT : acc = p.number1 ∗ p.number2; break;
8          case ACC_ADD :
9          ...
10         default :
11             // unknown op −> error response
12             p.calc_status = CALC_ERROR;
13     }
14     if (p.calc_status == CALC_OKAY) {
15         if (acc >= MAX_VAL || acc < MIN_VAL) {
16             p.calc_status = CALC_ERROR;
17             acc_out_of_range = true;
18         } else {
19             p.result = acc;
20         }
21     }
22 }
```

Fig. 1. Function *calculate*

2 Example

In the example (see also [8]), the development of a calculator is considered at two abstraction levels: the *behavioral level* and the *register transfer level*. The calculator shall be able to perform calculation (i.e. addition, subtraction, multiplication, etc.) with two given numbers. The calculator shall also be able to store the last calculated result and perform calculation with this number and another given number. A given number shall be an integer with up to 3 digits. If the result of a calculation has more than 3 digits, the calculator shall report an error.

2.1 Behavioral Level

First, a system description following a TLM modeling style is created. The data transported to and from the calculator is modeled as a payload containing the requested operator, two given numbers, and also the status and the result of the calculation. The functionality of the calculator shall be fully captured in a function *calculate* which receives a payload, performs the requested calculation, and writes back the result into the payload. The implementation of *calculate* is depicted in Fig. 1.

```
1   always @(alu_result, error)
2   begin
3     old_result = alu_result;
4     status = 1;
5     if ((old_result[32] == 1 && old_result > MIN_VALUE)
6         || (old_result[32] == 0 && old_result >= MAX_VALUE))
7     begin
8       status = 0;
9     end
10
11    if (error)
12    begin
13      old_result = 0;
14      status = 0;
15    end
16  end
```

Fig. 2. Part of the RTL model containing the bug

The SystemC implementation has passed all testcases and thus has been released for further development steps. However, the testcases have missed the bug on Line 15 ($>= MAX_VAL$ should be instead $> MAX_VAL$). This bug causes the calculator to report an error if the calculated result is MAX_VAL (999). Since this corner case has not been considered during testing, the bug propagates to the next abstraction level RTL.

2.2 Register Transfer Level

The RTL model is created in a refinement process starting with the behavioral SystemC model. First, the payload is refined to inputs and outputs of the overall design: both numbers and the operator become inputs, while the result and the calculation status become outputs. The function *calculate* of the SystemC model is refined to two additional modules: the module CALCULATE to perform the actual calculation, and the module SELECT that stores the last calculated result and delivers it to CALCULATE when an accumulative operation is chosen. Two existing IP components from the M1 Core [9] have been integrated into the module CALCULATE: an *Arithmetic Logic Unit* (ALU) – for the addition and subtraction – and a multiplier. Some parts of the algorithmic behavior can be translated one-to-one, for example, the range check, and thus the described bug is still present. This can be observed in Fig. 2 where Line 6 contains the same erroneous comparison.

The overall structure of the RTL design is depicted in Fig. 3. As can be seen, the two number inputs, the operator input, the result output, and the calculation status are denoted as a, b, op, $results$, and $status$, respectively. In each calculation, the inputs are checked first in the unit CHK_1 whether they are within the valid range. Then, they are forwarded to the CALCULATE module.

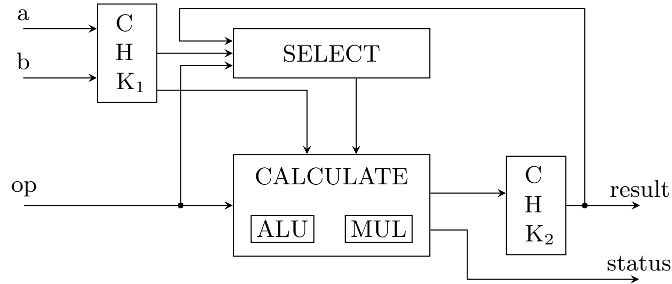

Fig. 3. RTL model overview

The CALCULATE module calculates the result using either the ALU or the multiplier depending on the value of the input *op*. This result is then checked again in unit CHK_2.

2.3 Synchronized Debugging

After the RTL model has been completely implemented, its correctness has also to be verified. For this task, a set of RTL properties has been written and formally checked using WoLFram [10] During this phase, the bug described above has been detected and also a counter-example has been delivered.

Now, using automated debugging methods for RTL (e.g. [11,12,13]), the bug at Line 6 in Fig. 2 can be identified among a list of candidates and fixed. Synchronized debugging goes a step further: First, the counter-example at RTL is abstracted to create a new testcase at the behavioral level. This testcase fails and thus confirms the existence of the same bug in the behavioral model. Instead of applying automated debugging approaches for SystemC TLM (e.g. [14,15]), the debugging result at RTL can be reused. Because the correspondence between the elements of each respective abstraction level is kept, we know that Line 15 in Fig. 1 corresponds to Line 6 in Fig. 2. Thus, it is possible to determine Line 15 in Fig. 1 as the location of the bug.

3 Conclusions

In this paper we have introduced the concept of *synchronized debugging* and demonstrated its usefulness on a simple example. The main advantage of this concept is that once a bug has been found, its location and fix can be propagated across all abstraction levels. Thus, models being used at different levels of abstraction are always fixed and kept in sync with minimum debugging effort. The key enabler for the methodology is the correspondence between the models of two different abstraction levels. However, in practice, it is not always as simple as in our example to establish this correspondence during the refinement steps. Hence, future research will mainly focus on this open question, which is not only important for synchronized debugging but also for e.g. equivalence checking.

References

1. Wilson Research Group and Mentor Graphics: 2010-2011 Functional Verification Study (2011)
2. Bailey, B., Martin, G., Piziali, A.: ESL Design and Verification: A Prescription for Electronic System Level Methodology. Morgan Kaufmann/Elsevier (2007)
3. Ghenassia, F.: Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems. Springer (2006)
4. Accellera Systems Initiative: SystemC (2012) Available at www.systemc.org.
5. Black, D.C., Donovan, J.: SystemC: From the Ground Up. Springer-Verlag New York, Inc. (2005)
6. Große, D., Drechsler, R.: Quality-Driven SystemC Design. Springer (2010)
7. Aynsley, J.: OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL. Open SystemC Initiative (OSCI) (2009)
8. Drechsler, R., Diepenbeck, M., Große, D., Kühne, U., Le, H.M., Seiter, J., Soeken, M., Wille, R.: Completeness-driven development. In: International Conference on Graph Transformation. (2012) 38–50
9. Fazzino, F., Watson, A.: M1 core (2012) Available at http://opencores.org/project,m1_core.
10. Sülflow, A., Kühne, U., Fey, G., Große, D., Drechsler, R.: WoLFram - a word level framework for formal verification. In: IEEE/IFIP International Symposium on Rapid System Prototyping. (2009) 11–17
11. Fey, G., Staber, S., Bloem, R., Drechsler, R.: Automatic fault localization for property checking. IEEE Trans. on CAD **27**(6) (2008) 1138–1149
12. Große, D., Fey, G., Drechsler, R.: Enhanced formal verification flow for circuits integrating debugging and coverage analysis. In R. Ubar, J. Raik, H.T.V., ed.: Design and Test Technology for Dependable Systems-on-Chip. Information Science Reference (2011) 119–129
13. Debug!t - Automated Debugging for Chip Design. `www.solvertec.de`. (2013)
14. Le, H.M., Große, D., Drechsler, R.: Automatic TLM fault localization for SystemC. IEEE Trans. on CAD **31**(8) (August 2012) 1249–1262
15. Le, H.M., Große, D., Drechsler, R.: Scalable fault localization for SystemC TLM designs. In: Design, Automation and Test in Europe. (2013)