Completeness-Driven Development[⋆]

Rolf Drechsler[1,2], Melanie Diepenbeck[1], Daniel Große[1], Ulrich Kühne[1], Hoang M. Le[1], Julia Seiter[1], Mathias Soeken[1,2], and Robert Wille[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. Due to the steadily increasing complexity, the design of embedded systems faces serious challenges. To meet these challenges additional abstraction levels have been added to the conventional design flow resulting in *Electronic System Level* (ESL) design. Besides abstraction, the focus in ESL during the development of a system moves from design to verification, i.e. checking whether or not the system works as intended becomes more and more important. However, at each abstraction level only the validity of certain properties is checked. Completeness, i.e. checking whether or not the entire behavior of the design has been verified, is usually not continuously checked. As a result, bugs may be found very late causing expensive iterations across several abstraction levels. This delays the finalization of the embedded system significantly. In this work, we present the concept of *Completeness-Driven Development* (CDD). Based on suitable completeness measures, CDD ensures that the next step in the design process can only be entered if completeness at the current abstraction level has been achieved. This leads to an early detection of bugs and accelerates the whole design process. The application of CDD is illustrated by means of an example.

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. To keep up with the (customer) requirements, design reuse is common and, hence, more and more complex *Intellectual Property* (IP) is integrated. According to a recent study [1], the external IP adoption increased by 69% from 2007 to 2010. In 2010, 76% of all designs included at least one embedded processor. As a result, the development of embedded systems moves from design to verification, i.e. more time is spent in checking whether the developed design is correct or not. In fact, in the above mentioned time period, 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.

[⋆] This work was supported in part by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1.

A major milestone for the development and verification of embedded systems has 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, 4]. As modeling language typically SystemC [5–7] is used which offers the TLM-2.0 standard [8]. 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.

While this flow is established in industry today, ESL-based design focuses on the implementation and verification of the system. However, although the validity of certain properties of the implementation is checked at the various abstraction levels, often the behavior is not completely considered in these stages. Completeness, i.e. checking whether or not certain behavior of the resulting design has been verified, is usually not continuously checked. This typically causes expensive iterations across several abstraction levels and delays the finalization of the embedded system significantly.

In this work, we present the concept of *Completeness-Driven Development* (CDD). The idea of CDD ensures that the next step in the design process can only be entered if completeness at the current abstraction level has been achieved. For this purpose, suitable completeness measures are needed for each abstraction level in a CDD flow. With CDD, the focus moves from *implementation* to *completeness* while completeness is targeted immediately. Overall, CDD has the following advantages:

- **In-place verification:** New details are verified when they are added.
- **No bug propagation:** Bugs are found as soon as possible since completeness ensures verification quality. As a consequence, bugs are not propagated to lower levels.
- **Long loop minimization:** Loops over several abstraction levels may only occur due to design exploration or unsatisfied non-functional requirements.
- **Correctness and efficient iterations:** The essential criterion is design correctness which is ensured via completeness along each design step. Thus, iterations are only necessary at the current abstraction level.

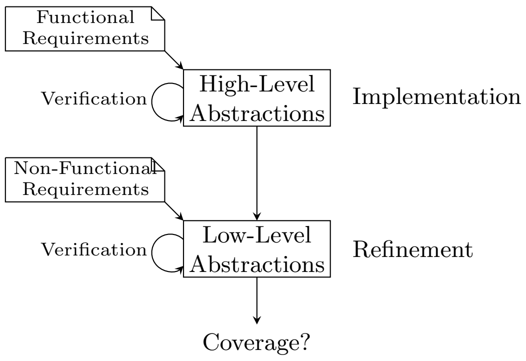

Fig. 1. Conventional flow

CDD is illustrated by means of an example at two representative abstraction levels (Sect. 3). Before, the addressed problem and the proposed CDD flow is described in more detail in the next section.

2 Completeness-Driven Development

2.1 Established ESL Flow

Figure 1 shows a rough sketch of the established ESL flow for embedded systems. To cope with the increasing complexity, requirements for the system are not incorporated at once, but subsequently added leading to a continuous refinement. Usually, the functional requirements are considered first at higher levels of abstraction. Non-functional requirements are added afterwards in the lower levels of the design flow. This allows designers to concentrate on the behavior of the system first. This procedure is sufficient for early simulation (through an executable specification) as well as analysis of the correctness of the functional aspects.

As can be seen in Fig. 1, newly incorporated requirements are verified against prior design states and the specification. However, although a positive verification outcome ensures the correctness of the system with respect to the properties that are checked, full correctness cannot be ensured as it is unclear whether enough properties have been considered. For this task, completeness checks have to be applied. However, today completeness checkers are typically not used continuously, i.e. coverage checks are performed after several design steps and, even worse, mainly at lower levels of abstraction – too late in the overall design process.

As a consequence, behavior that has not been verified at the current abstraction level is not considered until the lower stages of the design flow. If it turns out that this unconsidered part contains bugs, a large portion of the design flow needs to be repeated, leading to long and expensive verification and debugging loops.

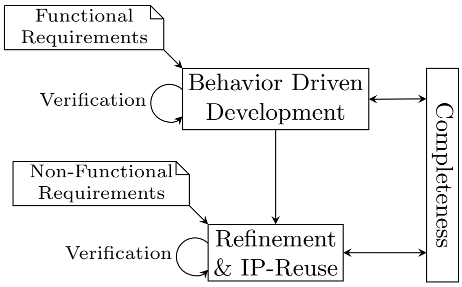

Fig. 2. Overview CDD flow

2.2 Envisioned Solution

To solve the problem described above, we propose the concept of CDD. CDD shifts the focus from implementation to completeness, i.e. completeness is added orthogonally to the state-of-the-art flow. By this, completeness checks are performed at a high level of abstraction and during all refinement steps. As a consequence in each situation, the next step in the design process can only be entered if completeness at the current abstraction level has been achieved.

A major requirement for this flow is that suitable coverage measures must be available for each abstraction level. For the lower levels of abstractions different approaches already exist, see e.g. [9–13]. Also solutions of industrial strength are available, for instance [14]. In contrast, on higher level of abstraction only a few approaches have been proposed. Most of them are based on simulation, e.g. [15–18], and, hence, are not sufficient since the identification of uncovered behavior is not guaranteed. On the formal side, initial approaches have been devised for instance in [19, 20]. If, within a specific abstraction level, an implementation step can be adequately formalized as a *model transformation*, then completeness results can be propagated through several transformations, as long as their correctness is ensured. As an example, in [21], behavior preserving transformations are used to refine the communication model of a system.

In the following, we demonstrate CDD at two representative abstraction levels. The design is composed through *Behavior Driven Development* (BDD) [22] and subsequent refinement/IP-reuse. BDD is a recent development approach which has its roots in software *Test Driven Development* (TDD) [23]. Essentially, in TDD testing and writing code is interleaved while the test cases are written before the code. In doing so, testing is no longer a post-development process, which in practice is often omitted due to strict time constraints. BDD extends TDD in the sense that the test cases are written in natural language, easing the communication between engineers and stakeholders. In BDD, all test cases are called *acceptance tests*. To summarize, in contrast to the current flow from Fig. 1, with CDD completeness is considered additionally at each abstraction level as shown on the right hand side in Fig. 2. This is demonstrated using a concrete example in the next section.

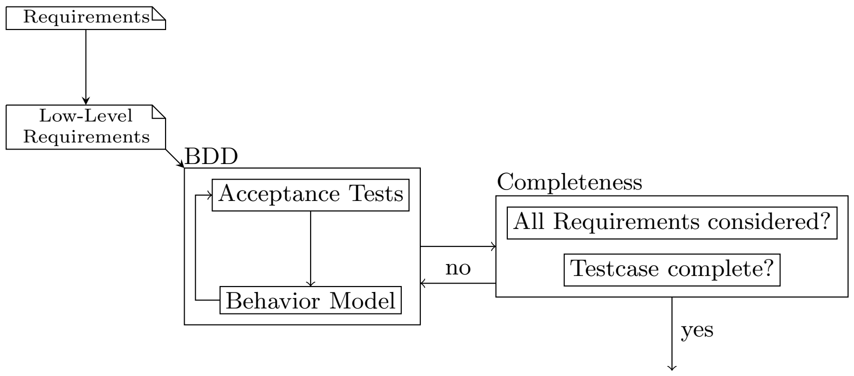

Fig. 3. CDD flow on high-level abstractions

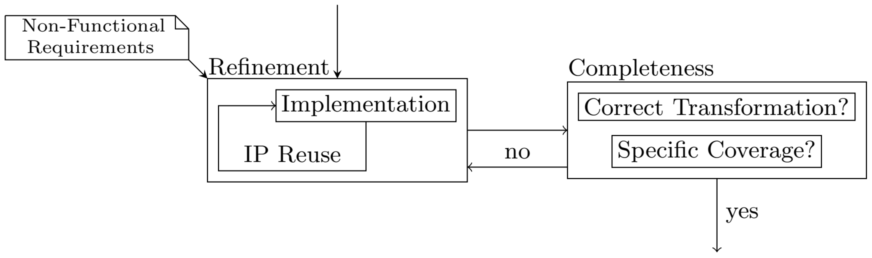

Fig. 4. CDD flow on lower-level abstractions

3 Completeness-Driven Development in Action

In this section, we present an example to demonstrate the proposed CDD flow. We first review the applied abstraction levels and the respective flows at a glance. The details are then explained in the following subsections.

In the example, the development of a calculator is considered. In the process, we use two abstraction levels: the *behavioral level* and the *register transfer level*. The overall flow from Fig. 2 is partitioned into two subflows for each respective abstraction level as depicted in Figs. 3 and 4, respectively. As can be seen, in conjunction with BDD at the behavioral level and the refinement/IP-reuse approach at RTL, completeness analysis techniques tailored for each respective method are employed (see right hand side of both figures).

The development process starts with a document of requirements (see Fig. 3). The translation of these initial requirements to acceptance tests requires an intermediate step that derives low-level requirements. The following BDD process first produces the acceptance tests from these requirements and generates a set of corresponding testcases written in SystemC. Afterwards, the SystemC behavioral model is incrementally developed to pass the defined testcases one at a time. The defined behavior has been fully implemented, if the model passes all testcases. As a consequence, we perform a completeness check to ensure that all requirements have already been considered and the testcases are complete. The result of this step is a complete set of properties that have been generalized from the tests.

After the completeness at behavioral level has been achieved, we proceed to the lower abstraction level at RTL (see Fig. 4). The SystemC model is refined to an RTL model in Verilog. In this refinement step, IP components are integrated. The functionality and the completeness of the RTL model are subsequently assured by using property checking and property-based coverage analysis, respectively.

In the remainder of this section, we first briefly describe the path from the initial requirements to the acceptance tests of BDD. Afterwards, the model and the development process at the behavior level and the RTL are presented in necessary detail focusing on the completeness analysis at both levels of abstractions.

3.1 From Requirements to Acceptance Tests

An excerpt of a list of requirements describing the functionality of the considered calculator reads as follows:

REQ1. The system shall be able to perform calculation with two given numbers. At least addition, subtraction, and multiplication shall be supported.

REQ2. The system shall be able to store the last calculated result and perform calculation with this number and another given number.

REQ3. A given number:
1. can have a positive integer value;
2. can have a negative integer value;
3. shall have up to 3 digits;
4. can be 0.

REQ4. If the result of a calculation has more than 3 digits, the system shall report an error.

REQ...

These requirements are then translated to low-level requirements that capture precisely the expected behavior of the calculator in each specific case. For example, the first two low-level requirements (LLR1 and LLR2 in the following) specify the addition operation of the calculator. Note that the relation to the initial high-level requirements is maintained when specifying each low-level requirement.

LLR1. The system shall be able to add two given numbers in the range $[-999, 999]$. If the sum of the given numbers fits in the range $[-999, 999]$, the system shall return this value. This requirement corresponds to REQ1, REQ3 and REQ4.

LLR2. The system shall be able to add two given numbers in the range $[-999, 999]$. If the sum of the given numbers does not fit in the range $[-999, 999]$, the system shall report an error. This requirement corresponds to REQ1, REQ3 and REQ4.

LLR3. The system shall be able to subtract the second given number from the first one ...
LLR...

At the beginning of the BDD process, the low-level requirements are compiled into acceptance tests which are provided in a very close form to a testcase and also contain precise values of the numbers given to the calculator. For example, the acceptance test that corresponds to LLR1 is as follows:

When the numbers $< a >$ and $< b >$ are given
And I want to add $< a >$ and $< b >$
Then the result should be $< c >$ (where $< c >=< a > + < b >$)
Examples:

a	0	7	20	1	...
b	0	4	17	997	...
c	0	11	37	998	...

Through BDD, these acceptance tests can now be used to determine an according SystemC description.

3.2 CDD at High Level of Abstraction

Generating the Behavioral Model in SystemC. First, the BDD process generates a system description following a TLM modeling style. That is, the data transported to and from the calculator is modeled as a payload shown in Fig. 5. It contains 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.

After this basic structure has been defined, the BDD process continues with the translation of the acceptance tests to executable testcases in SystemC. Figure 6 exemplarily shows a testcase which corresponds to one of the precise cases in the acceptance test for the low-level requirement LLR1 shown earlier. Line 2 declares a SystemC port to which the calculator will be connected later. In Lines 11–16, a payload with a request operator as well as numbers is initialized and sent to the calculator through the port, while afterwards the received results are checked.

After all testcases have been written, the SystemC model (essentially the function *calculate*) is developed step-by-step to gradually pass all testcases. The final version of *calculate* is depicted in Fig. 7. For example, the first development step has added Line 5 and Lines 14–21 to satisfy the testcases defined for the addition of two given numbers. Lines 14–21 check the intermediate result, then raise the error status flag or write the valid result back, respectively. This code lines are also common for the other operations, so that only Line 6 and Line 7 had to be added to make the testcases for subtraction and multiplication pass. The SystemC model has been successfully tested against all defined testcases.

```
struct calc_payload {
    Operator op;
    int number1;
    int number2;
    CalcStatus calc_status;
    int result;
};
```

Fig. 5. Calculator payload

```
struct testcase : public sc_module {
    sc_port<calculator_if> calc_port;

    SC_HAS_PROCESS(testcase);

    testcase(sc_module_name name) : sc_module(name) {
        SC_THREAD(main);
    }

    void main() {
        calc_payload p;
        p.op = ADD;
        p.number1 = 7;
        p.number2 = 4;
        calc_port->calculate(p);
        assert(p.calc_status == CALC_OKAY && p.result == 7+4);
    }
};
```

Fig. 6. A testcase for the calculator

Checking the Completeness. Since the relation between the initial requirements, the low-level requirements, the acceptance tests, and the testcases in SystemC have always been maintained in each translation step, it is very easy to trace back and check whether all requirements have been considered.

To check the completeness of the testcases, they are first generalized into formal properties. As the whole functionality of the calculator is captured in the function *calculate*, the properties only need to reason about the behavior at the start and the end of each *calculate* transaction. Most of the generalization process can be automated, however, human assistance is still required in providing adequate invariants.

To illustrate the concept, Fig. 8 shows two generalized properties for the addition of two given numbers. Both properties are written in a flavor of the *Property Specification Language* (PSL) extended for SystemC TLM [24]. As mentioned earlier, we only need to sample at the start (*:entry*) and the end (*:exit*) of the function/transaction *calculate*. Property P1 covers the case that the sum of two

```
void calculate(calc_payload& p) {
    p.calc_status = CALC_OKAY;
    switch (p.op) {
        case NOP : break;
        case ADD : acc = p.number1 + p.number2; break;
        case SUB : acc = p.number1 − p.number2; break;
        case MULT : acc = p.number1 ∗ p.number2; break;
        case ACC_ADD :
        ...
        default :
            // unknown op −> error response
            p.calc_status = CALC_ERROR;
    }
    if (p.calc_status == CALC_OKAY) {
        if (acc > MAX_VAL || acc < MIN_VAL) {
            p.calc_status = CALC_ERROR;
            acc_out_of_range = true;
        } else {
            p.result = acc;
        }
    }
}
```

Fig. 7. Function *calculate*

given numbers fits in the range so that the calculation will be successful and the sum will be returned, while P2 specifies the calculation in the other case, i.e. the sum is out of range. In both cases, the valid range had to be provided manually as an invariant. Both properties then represent the generalized behavior which is partly considered by the testcases. This generalized behavior is also proven by the high-level property checking method in [25].

However, as determined by the completeness check, behavior remained uncovered. In fact, the invariant for P1 is insufficient. More precisely, the completeness check has detected an uncovered testcase:

$$p.number1 == 0 \text{ and } p.number2 == 999.$$

This is representative for the general forgotten case of

$$p.number1 + p.number2 == 999.$$

The result in this case is not defined by neither P1 nor P2. If this uncovered testcase would have been included in the set of testcases from the beginning, it would have been impossible to provide the insufficient invariant for P1, since a generalized property must be compliant with the testcases it covers. This demonstrates clearly the usefulness and necessity of completeness at this high level of abstraction.

```
P1: default clock = calculate:entry || calculate:exit;
always (calculate:entry && p.op == ADD && (−999 < p.number1 +
          p.number2 && p.number1 + p.number2 < 999))
     −> next (calculate:exit && p.calc_status == CALC_OKAY &&
          p.result == p.number1 + p.number2)

P2: default clock = calculate:entry || calculate:exit;
always (calculate:entry && p.op == ADD && ((p.number1 +
          p.number2 >= 1000) || (p.number1 + p.number2 <= −1000))
     −> next (calculate:exit && p.calc_status == CALC_ERROR)
```

Fig. 8. Generalized properties for addition

3.3 CDD at RTL

Generating the RTL Model in Verilog. The RTL model is created in a refinement process starting with the behavioral SystemC model. First, the payload (see Fig. 5) 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. Subsequently, the sufficient bit-width for each input and output has to be determined based on the values it has to represent. Both number inputs and the result output are in the range $[-999, 999]$ and thus each of them needs 11 bits. The calculation status contains two states that can be represented using only one bit. For the operator input, three bits are required since its value can either be reset or one of the six supported arithmetic operators.

After the inputs and the outputs have been identified, we proceed to the translation of the algorithmic behavior. Some parts of the algorithmic behavior can be translated one-to-one, for example, the range check of the numbers. Before any computation, the respective inputs are checked if their values are within the valid range. 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.

To speed up the development, we integrate two existing IP components into the module CALCULATE: an *Arithmetic Logic Unit* (ALU) – for the addition and subtraction – and a multiplier. Both IPs are taken from the M1 Core [26]. The ALU itself has 15 different operation modes, including the arithmetic functions addition and subtraction, some shift and some Boolean operations. Both units can handle numbers up to 32 bits. Thus, an additional check has to be added to ensure that either the calculated result is in the allowed range or the status output is set.

The overall structure of the RTL design is depicted in Fig. 9. 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.

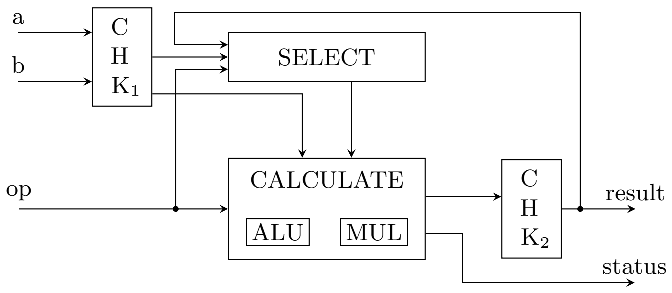

Fig. 9. RTL model overview

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.

Checking the Completeness. After the RTL model has been completely implemented, its correctness has to be verified. For this task, the complete set of properties at the behavioral model is also refined to a set of RTL properties. Essentially, timing needs to be added to the properties while adjusting the syntax of the PSL properties.

After all refined properties have been proven, we perform the completeness check at RTL using the method proposed in [13]. The check detects uncovered behavior of the RTL model for the value 111_2 of the 3-bit operator input *op*. The other seven values correspond to the defined operations of the calculator and hence the behavior in these cases is fully specified by the property set. In the case of 111_2, the ALU performs an unintended operation (a shift operation). This mismatch is possible because the ALU has not been specifically developed for the calculator (in fact as mentioned above the ALU is an external IP component). This shows clearly that completeness checks are necessary, in particular, since integrated IPs may have additional but unintended behavior.

4 Conclusions

In this paper, we have presented the concept of *Completeness-Driven Development* (CDD). With CDD, completeness checks are added orthogonally to the state-of-the-art design flow. As a result, completeness is ensured already at the highest level of abstraction and during all refinement steps. Hence, bugs are found as soon as possible and are not propagated to lower levels. As a result, expensive design loops are avoided. We have demonstrated the advantages of CDD for an example. For two abstraction levels (behavioral level and RTL) we have shown that completeness is essential for correctness and efficient development.

Going forward, to implement the concept of CDD, high-level and continuous completeness measures are necessary. Furthermore, innovative methods to support correct transformation as well as property refinement need to be investigated.

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. Cai, L., Gajski, D.: Transaction level modeling: an overview. In: IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp. 19–24 (2003)
4. Ghenassia, F.: Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems. Springer (2006)
5. Accellera Systems Initiative: SystemC (2012), `http://www.systemc.org`
6. Black, D.C., Donovan, J.: SystemC: From the Ground Up. Springer-Verlag New York, Inc. (2005)
7. Große, D., Drechsler, R.: Quality-Driven SystemC Design. Springer (2010)
8. Aynsley, J.: OSCI TLM-2.0 Language Reference Manual. Open SystemC Initiative (OSCI) (2009)
9. Chockler, H., Kupferman, O., Vardi, M.Y.: Coverage Metrics for Temporal Logic Model Checking. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 528–542. Springer, Heidelberg (2001)
10. Claessen, K.: A coverage analysis for safety property lists. In: Int'l Conf. on Formal Methods in CAD, pp. 139–145 (2007)
11. Große, D., Kühne, U., Drechsler, R.: Analyzing functional coverage in bounded model checking. IEEE Trans. on CAD 27(7), 1305–1314 (2008)
12. Chockler, H., Kroening, D., Purandare, M.: Coverage in interpolation-based model checking. In: Design Automation Conf., pp. 182–187 (2010)
13. Haedicke, F., Große, D., Drechsler, R.: A guiding coverage metric for formal verification. In: Design, Automation and Test in Europe, pp. 617–622 (2012)
14. Bormann, J., Beyer, S., Maggiore, A., Siegel, M., Skalberg, S., Blackmore, T., Bruno, F.: Complete formal verification of Tricore2 and other processors. In: Design and Verification Conference, DVCon (2007)
15. Helmstetter, C., Maraninchi, F., Maillet-Contoz, L.: Full simulation coverage for SystemC transaction-level models of systems-on-a-chip. Formal Methods in System Design 35(2), 152–189 (2009)
16. Heckeler, P., Behrend, J., Kropf, T., Ruf, J., Weiss, R., Rosenstiel, W.: State-based coverage analysis and UML-driven equivalence checking for C++ state machines. In: FM+AM 2010. Lecture Notes in Informatics, vol. P-179, pp. 49–62 (September 2010)
17. Bombieri, N., Fummi, F., Pravadelli, G., Hampton, M., Letombe, F.: Functional qualification of TLM verification. In: Design, Automation and Test in Europe, pp. 190–195 (2009)
18. Sen, A.: Concurrency-oriented verification and coverage of system-level designs. ACM Trans. Design Autom. Electr. Syst. 16(4), 37 (2011)
19. Apvrille, L.: Ttool for diplodocus: an environment for design space exploration. In: Proceedings of the 8th International Conference on New Technologies in Distributed Systems, NOTERE 2008 (2008)
20. Le, H.M., Große, D., Drechsler, R.: Towards analyzing functional coverage in SystemC TLM property checking. In: IEEE International High Level Design Validation and Test Workshop, pp. 67–74 (2010)

21. Andova, S., van den Brand, M.G.J., Engelen, L.: Reusable and Correct Endogenous Model Transformations. In: Hu, Z., de Lara, J. (eds.) ICMT 2012. LNCS, vol. 7307, pp. 72–88. Springer, Heidelberg (2012)
22. North, D.: Behavior Modification: The evolution of behavior-driven development. Better Software 8(3) (2006)
23. Beck, K.: Test Driven Development: By Example. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
24. Tabakov, D., Vardi, M., Kamhi, G., Singerman, E.: A temporal language for SystemC. In: Int'l Conf. on Formal Methods in CAD, pp. 1–9 (2008)
25. Große, D., Le, H.M., Drechsler, R.: Proving transaction and system-level properties of untimed SystemC TLM designs. In: ACM & IEEE International Conference on Formal Methods and Models for Codesign, pp. 113–122 (2010)
26. Fazzino, F., Watson, A.: M1 core (2012), http://opencores.org/project,m1_core