From Requirements and Scenarios to ESL Design in SystemC

Hoang M. Le[1] Daniel Große[1] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{hle, grosse, drechsle}@informatik.uni-bremen.de

***Abstract*—In the ESL design flow, the crucial task of developing a golden model that correctly implements the natural-language top-level specification has received little attention so far. The major drawback of the current practice is the isolation of design and verification. Motivated by this and the recent advance of verification techniques for SystemC ESL models, we propose a novel methodology to develop a correct SystemC golden model from the top-level specification. The proposed methodology is driven by the requirements and the scenarios in the specification with design and verification going hand in hand. An early formalization of requirements and scenarios produces a set of properties and a testbench together with a code skeleton that will be successively extended to a full SystemC ESL model. The availability of properties and a testbench beforehand enables verification-driven development of the model. The advantages of the methodology are discussed and demonstrated by a case study.**

I. Introduction

Today's *System-on-Chips* (SoCs) integrate an increasingly large number of hardware and software components. Such complex systems are extremely challenging to develop under tight time-to-market constraints. To cope with this very challenging development task, the level of abstraction has been raised beyond RTL to the so-called *Electronic System Level* (ESL) [1]. Communication and synchronization at ESL are modeled in terms of abstract operations and events rather than as low level signals or wires. For the description of ESL designs, the C++ class library *SystemC* [2], [3], [4] has become the de-facto standard. SystemC provides fundamental modeling components such as processes, modules, ports, interfaces and channels together with an event-driven simulation kernel to describe systems at different levels of abstraction. For abstracting the communication *Transaction Level Modeling* (TLM) has been standardized in SystemC [2].

The starting point of the ESL design flow is a top-level specification. This specification is written in natural language and contains requirements and scenarios for the system to be developed, which have been agreed upon by all the stakeholders. In [1], the ESL design flow is divided into six main steps in a top-down manner: specification and modeling, pre-partitioning analysis, partitioning, post-partitioning analysis and debug, post-partitioning verification, and HW/SW implementation. The top-level specification is manually converted to an ESL design in the first step. This first abstract design is considered a *golden model*. It is used as reference for the subsequent steps following the SystemC refinement methodology to create synthesizable hardware and software descriptions. However, the crucial task of developing a golden model, which correctly implements the natural language specification, has received little attention so far.

In current practice, the designer implements the first ESL model according to his interpretation of the specification. Then, the verification engineer formulates properties and testcases following his understanding of the specification and the already existing implementation. The major drawback here is the isolation of design and verification. The design might have been implemented in a way that makes it very challenging to verify (e.g. it is hard to formulate properties and/or to check them efficiently). Moreover, ambiguities and misinterpretations of the specification are detected rather late.

On the other hand, a lot of efforts have been made recently in verification of SystemC ESL designs: [5] has introduced a high-level property language for SystemC based on the *Property Specification Language* (PSL) [6]. Simulation-based [7], [8], [9], [10] and formal verification techniques [11], [12], [13] have been proposed. Debugging methods [14], [15], functional coverage [16], [17], and mutation-based coverage [18], [19], [10] have been also investigated.

To overcome the deficiencies of the current practice we propose a novel methodology to develop a correct SystemC golden model from the top-level specification. The proposed methodology is driven by the requirements and the scenarios in the natural language specification with design and verification going hand in hand. An early formalization of requirements and scenarios produces a set of properties and a testbench (a set of testcases) together with a SystemC code skeleton that will be successively extended to a full ESL model. The availability of properties and testbench beforehand enables verification-driven development of the model, i.e. the model is checked for correctness after some small changes using the mentioned ESL verification approaches.

The proposed methodology provides the following advantages:

- Early identification of ambiguities and inaccuracies in the specification (before implementation).
- Early bug detection and therefore the subsequent debugging process is simpler.
- Increased confidence of the model correctness since each modification can be immediately verified.
- More efficient verification because the code skeleton is created in sync with the formalized requirements.

We illustrate the methodology and the advantages for a concrete example, i.e. the development of a vending machine. As a result we obtained a correct golden model for the vending machine fully compliant with the specification which has been refined during the development process.

The remainder of this paper is structured as follows: In Section II related work is discussed. Then, Section III presents

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SANITAS under contract no. 01M3088 and by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1.

the proposed methodology. A case study demonstrating our approach is described in Section IV. Finally, Section V concludes the paper.

II. Related Work

The basic idea of integrating design and verification has been studied in the software domain. *Test-driven software development* (TDD) [20] is a software development approach where tests are written before the production code is created. Initially, the tests should fail. Then, the programmer writes production code to make the tests pass. Afterwards, the code is refactored until all defined tests pass again. TDD falls into the category of agile software development and step-wise improves the implementation.

An alternative software development approach is *Design-by-Contract* (DbC) [21]. In DbC the software developer defines formal interface specifications using pre-conditions, post-conditions and invariants forming the contracts. In this way, DbC is a process in the software design phase where the requirements are mapped into the software as contracts. There are many languages which natively support DbC (e.g. Eiffel and D) and various tools that enable DbC for standard languages.

A combination of TDD and DbC is *Specification-Driven Development* (SDD) proposed in [22]. In SDD it is not necessary to choose between the two approaches a priori. Instead, they are used together. For example, contracts can help when formulating tests and can act as test amplifier.

In *Property-Driven software Development* (PDD) [23] the specification, tests and executable model of the software are built together. PDD is an iterative process guided by user stories and use cases. After all tests succeed system properties are generalized and checked using simulation or model checking.

In summary, the briefly described software development approaches help to improve the software quality. However, they target the software development process and not ESL design.

For the automatic generation of behavior models in software development, formalized scenarios (e.g. UML sequence diagrams in [24]) and formalized requirements (e.g. Object Constraint Language pre- and postconditions in [24], Fluent Linear Temporal Logic properties in [25]) have been used.

In [26], formal specification is utilized to improve the development process of software drivers and hardware devices. Based on the English specification, three models for software, hardware and HW/SW interface, respectively, are developed in a C-based specification language. However, the focus of the paper is on the added value of these models in the development process. The process to create them starting from the specification is not elaborated.

On the pure hardware side in the PROSYD project [27] a property-based design flow has been developed. It includes requirement definition, design, implementation, and verification and is built on PSL. However, the focus of the approach is clearly on hardware and only lower levels of abstraction, i.e. RTL, are considered.

III. Methodology

The proposed methodology is introduced in this section. The overall flow is depicted in Fig. 1. As can be seen, the flow starts with the top-level specification written in natural language. This document has served as the communication mechanism between the stakeholders and has been agreed upon by all. The specification consists of requirements and

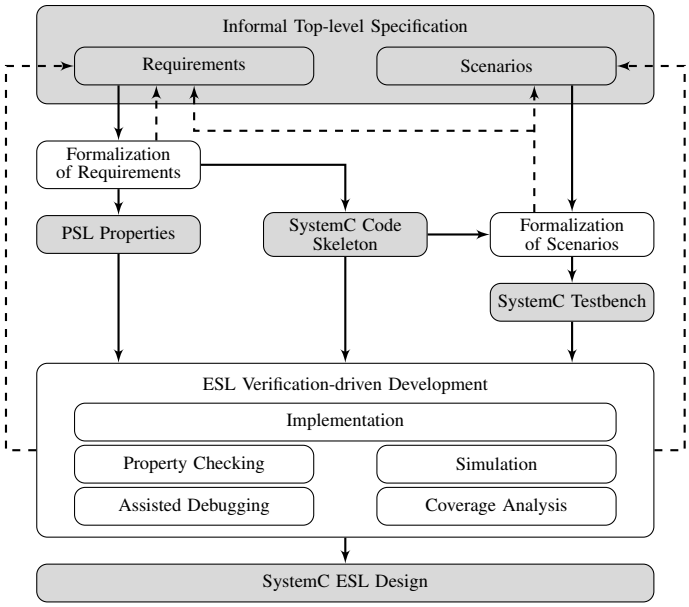

Fig. 1. Overall Design Flow

scenarios. The general behavior of the system to be developed is captured by the requirements, while the scenarios describe the observable behavior in concrete cases (i.e. given inputs and expected outputs). Actually, the requirements alone can fully describe the behavior but they are generally harder to grasp and communicate. The scenarios complement them in this aspect.

The first step of our methodology is the *formalization* of the specification. This step is divided further in two substeps:

1) First, the requirements are translated into a set of high-level properties in PSL. The behavior described by the requirements is formalized as sequences of abstract operations controlled by conditions concerning abstract data. The defined operations and data should also be found in the design, otherwise the properties cannot be checked. Thus, this formalization step also produces a code skeleton in SystemC containing these elements. Furthermore, the formalization requires to carefully think about the natural language description, and thus also helps resolving ambiguities and inaccuracies in the requirements. Once detected, these deficiencies should be fixed before the formalization is continued.
2) Then, the scenarios are translated into SystemC testcases based on the existing code skeleton. The translation also helps resolving ambiguities and inaccuracies in the scenarios. Given inputs and their expected outputs are linked together with the defined operations and data. Inconsistencies between the scenarios and the requirements and missing requirements can be identified as well - then the translation of a scenario is not possible, because for instance some needed data has not been defined yet in the code skeleton. In such a situation the requirements need to be updated and the first substep has to be repeated.

After the formalization has been done, we have a set of PSL properties, a testbench (a set of SystemC testcases), and a SystemC code skeleton. At the beginning of the *development step*, the code skeleton can already be checked against the properties and the testbench, but the results are negative. The development is driven by the verification results: the code

TABLE I
FORMALIZATION OF REQUIREMENTS

REQ	Textual description	
	Term_i	Formalization_i
	PSL Property	

REQ1	The user can choose any item that is available.	
	Availability of an item X	Integer data $quantity[X]$, a positive value means X is available
	Choose an item X	Operation $choose(X)$, returns "true" if successful, "false" otherwise
	Property PSL1: *default clock = choose:**exit**;* ***always*** *((quantity[choose:1] > 0)−>choose:0)*	
REQ2	An item will only be released, if at least the price of the item has been paid.	
	The amount of money already paid	Integer data $paid\_amount$
	Price of an item X	Integer data $price[X]$
	Release an item X	Operation $release\_item(X)$
	Property PSL2: *default clock = release_item:**entry**;* ***always*** *(paid_amount >= price[release_item:1])*	
REQ3	As soon as the user pays the exact price of the chosen item, the item will be released.	
	The chosen item	Integer data $chosen\_item$
	Pay an amount Z	Operation $pay(Z)$
	Property PSL3: *default clock = ∗:**exit**; //sample at the end of any operation* ***always*** *((pay:**exit** && paid_amount == price[chosen_item])* *−>**next** (release_item:**exit** && release_item:1 == chosen_item))*	
REQ4	The vending machine can only release an item which is available	
	Property PSL4: *default clock = release_item:**entry**;* ***always*** *(quantity[release_item:1] > 0)*	
REQ...	...	

skeleton is incrementally extended to a full model so that more and more properties and testcases become successfully evaluated. Ideally, in each development step, the developer chooses either an unsatisfied property or a failed testcase, and subsequently extends the code skeleton so that the functionality described by the chosen property/testcase is fulfilled. As mentioned in the introduction, recently proposed ESL verification approaches for SystemC can be employed in this process. If a code extension violates a property or a testcase that passed before, debugging techniques are applied to find and fix the bug(s). After the property set and the testbench have been completely satisfied, coverage analysis is employed. This enables the detection of scenarios that are covered by neither the property set nor the testcases. If such a scenario is detected, the specification needs to be extended to capture it (see dashed arrows on the left and right in Fig. 1). This incremental change to the specification is then propagated through the formalization step down to development.

At the end of the development step, a SystemC ESL model fully compliant with the formalized specification (i.e. the PSL property set and the SystemC testbench) results. The compliance with the top-level specification still depends on the formalization step. Although most ambiguities can be identified and fixed in this step, no absolute guarantee can be given. Therefore, it is also very important that the formalization step and the development step are not solely performed by one designer/engineer/team.

In the next section we demonstrate the proposed methodology for a concrete example.

IV. CASE STUDY

In the case study we consider the development of a vending machine. First, we briefly describe the property language used in the formalization. Then, requirements of the vending machine in English and their formalization as properties are given. Afterwards, the code skeleton produced by this formalization is discussed. Next, we show how testcases can be developed from the scenarios described in the specification. Finally, we discuss the verification-driven development of the vending machine.

A. High-level Properties at ESL

For property specification an extension of PSL [6] is adopted. This extension [5] has introduced additional primitives – coming from the software world – which are well suited for ESL property specification. Besides the variables in the design, the following primitives are used:

- *func_name:**entry*** - start of a function/operation
- *func_name:**exit*** - end of a function/operation
- *event_name:**notified*** - notification of an event
- *func_name:**number*** - return value in case $number = 0$ and parameters of a function/operation otherwise

As *default temporal resolution* for evaluating the temporal operators we sample at all *system events*, which is either the start or the end of any abstract operation or the notification of any event. It is possible to change the temporal resolution, e.g. to sample only at notification of a certain event. For details on the used property checking approach we refer to our previous work [11].

B. Requirements

Generally, for the formalization of a requirement we need to break down the requirement into individual terms and formalize these first. The individual formal elements are then combined to create properties. Table I presents the textual description, the terms, their formalization and the final PSL property for an excerpt of the initial requirements. In the following we exemplarily describe the process in more detail for three requirements.

REQ2: An item will only be released, if at least the price of the item has been paid.

For the formalization of the first part, we need to define the term "to release an item". This term describes clearly an action and thus is defined as an abstract operation $release\_item$. For the second part, two terms need to be formalized: "the price of an item" and "the amount that has been paid". Both terms should apparently be defined as abstract integer data: $price[X]$ for an item X and $paid\_amount$, respectively. Now, the requirement can be formulated as the property PSL2 shown in Table I, which reads as follows: everytime an item should be released (*release_item:**entry***), the paid

amount should be at least the price of the item to be released (*price[release_item:1]*, the item to be released corresponds to the first argument of the function *release_item*).

REQ3: As soon as the user pays the exact price of the chosen item, the item will be released.

This requirement looks very similar to REQ2, but there is a substantial difference: REQ3 describes a temporal sequence of actions, while REQ2 only specifies the condition to release an item. The action "to release an item" has been defined already. The second action "to pay" is now formalized as an abstract operation pay. As the term "exact price of an item" has also been defined, we only need to define "the chosen item" to complete the formalization of REQ3. This term is defined as abstract integer data $chosen\_item$, respectively. Now, the requirement can be formulated as the property PSL3 shown in Table I. which reads as follows: after the user has paid some amount (*pay:**exit***) and if the paid amount has become equal to the price of the chosen item, the next operation of the vending machine is to release this item (this corresponds to the term "as soon as" in the requirement).

REQ4: The vending machine can only release an item which is available.

As can be seen, both terms needed to formalize this requirement are already defined: "to release an item" and "an item is available" (item X is available if the value of the abstract data $quantity[X]$ is positive, see REQ1 in Table I). The requirement is translated to the property PSL4 in a similar manner to REQ2.

C. Code Skeleton

The formalization of the requirements produces the code skeleton depicted in Fig. 2. This step has not been automated in the case study. As can be seen from Fig. 2, the defined abstract operations are contained in a *sc_interface* (see Line 1-7). The vending machine extends this *sc_interface* and thus should implement all operations. Furthermore, all defined abstract data can be seen in the definition of the struct *vending_machine* (see Line 9-13). The code skeleton lays the basis for the next step where we consider the scenarios as given in the top-level specification.

D. Scenarios

This section describes how the scenarios of the vending machine specification are translated into testcases. As mentioned before, the translation is also based on the code skeleton described in the previous section. In the following we exemplarily give the description of some scenarios and their translation.

Before that, we define the common structure for all testcases captured by the abstract testcase shown in Fig. 3. As can be seen this testcase has a *sc_port* connected to a vending machine instance, and a *SC_THREAD main()* which first executes *setup()* to setup the availability and the price of the items and then the virtual method *test*. This virtual method is to be defined for each scenario individually. Note that the vending machine interface (see Fig. 2) does not provide a mechanism for the setup. Thus, we need to extend the interface by a method *setup_item*, which set the *quantity* and *price* for each item accordingly. Note that this method is only needed to setup the testcase so it does not correspond to a missing requirement. Now, we describe two scenarios.

```
1  class vending_machine_if : public sc_interface {
2  public:
3    virtual bool choose(unsigned) = 0;
4    virtual void release_item(unsigned) = 0;
5    virtual void pay(unsigned) = 0;
6    ...
7  };
8  struct vending_machine : public sc_module, public
         vending_machine_if {
9    unsigned quantity[MAX_ITEM];
10   unsigned price[MAX_ITEM];
11   unsigned paid_amount;
12   unsigned chosen_item;
13   bool change_available;
14   vending_machine(sc_module_name name)
15   : sc_module(name) { // to be implemented }
16   // to be implemented
17 };
```

Fig. 2. Code skeleton produced by formalization of requirements

```
1  struct testcase : public sc_module {
2    sc_port<vending_machine_if> cm_port;
3    SC_HAS_PROCESS(testcase);
4    testcase(sc_module_name name)
5    : sc_module(name) { SC_THREAD(main); }
6
7    void setup() {
8      // setup quantity and price
9      cm_port->setup_item(5, 3, 90); // item 5, quantity 3, price
           0.90 EURO
10     cm_port->setup_item(6, 0, 50);
11     cm_port->setup_item(12, 1, 120);
12     cm_port->setup_item(14, 2, 80);
13     cm_port->setup_item(16, 4, 100);
14   }
15   virtual void test() = 0;
16   void main() { setup(); test(); }
17 };
```

Fig. 3. Abstract testcase

SCE1: An user approaches the vending machine. The machine is ready. The user chooses item number 12. The user pays exactly the price of this item. The machine releases a piece of item 12. Afterwards it is ready for the next purchase.

As can be seen, this scenario requires the vending machine to be ready for purchase at the beginning and the end of the scenario. This cannot be formulated on the code skeleton yet, since currently the vending machine has no status. Furthermore, it is clearly a functional requirement that that the vending machine should be ready again after a purchase. Thus, the formulation of this scenario has detected this missing requirement. After the requirements have been extended, a data field indicating the current status of the vending machine is added to the code skeleton. A function *is_ready()* is also added to allow the testcase to query this status. To check if the machine has released a piece of item 12, we also need to define the function *get_released_item()*. Overall, the scenario SCE1 can then be formulated as the following testcase:

```
1  struct testcase1 : public testcase {
2    void test() {
3      assert(cm_port->is_ready());
4      cm_port->choose(12);
5      wait(10, SC_NS);
6      cm_port->pay(120);
7      wait(10, SC_NS);
8      assert(cm_port->get_released_item() == 12);
9      assert(cm_port->is_ready());
10   }
11 };
```

Note that the *wait()* statements are not part of SCE1, but needed for the SystemC simulation. Now, the second scenario is described.

SCE2: An user approaches the vending machine. The machine is ready. The user chooses item number 14. The user

throws in a 2 EURO coin. The machine releases a piece of item 14 and 1.20 EURO change. Afterwards it is ready for the next purchase.

This scenario is very similar to the last one SCE1, except that the user pays more than the price of the wanted item. To check the returned amount, we need to add the corresponding function to the vending machine implementation. The testcase is shown in the following code:

```
1  struct testcase2 : public testcase {
2    void test() {
3      assert(cm_port->is_ready());
4      cm_port->choose(14);
5      wait(10, SC_NS);
6      cm_port->pay(200);
7      wait(10, SC_NS);
8      assert(cm_port->get_released_item() == 14);
9      assert(cm_port->get_returned_amount() == 120);
10     assert(cm_port->is_ready());
11   }
12 };
```

E. Verification-driven Development

The vending machine has been extended from the initial code skeleton to a full ESL SystemC design. The testbench has been successfully simulated on this design (i.e. all testcases have passed). The defined properties have been verified very fast using the approach in [11]. One of the main reasons for the efficiency of the verification can be explained as follows. Due to the early formalization, the properties have been formulated without restrictions posed by an existing implementation. Thus, the properties could be kept very simple. As can be seen in Table I, all properties belong to the *simple subset* of PSL, which is known to be much easier to verify [6]. The properties that have been formulated later for the detected missing requirements also belong to this class.

In the following we discuss one of the bugs encountered during the development process in more detail. After the defined operations *choose*, *pay*, and *release_item* have been implemented, we checked the property set on the partial model and discovered a violation of the property PSL4. The violating sequence provided by the approach in [11] is as follows: the user chooses an item and pays its price, the vending machine releases the last available piece of this item, then the user pays the exact price once again, finally the vending machine releases one more piece of the item. The last action is clearly not possible because the last piece has been already released. This bug reveals an inaccuracy in the specification: the requirement specifying the behavior after a purchase is missing. Concretely, after the current purchase is finished, the current chosen item needs to be invalidated so that the user needs to choose an item again for the next purchase, and thus the violating sequence is prevented. After adding this requirement to the specification, we formulated the corresponding property and extended the model for this behavior. After that, the newly added property as well as the property PSL4 became satisfied and the development process could be continued.

V. Conclusions

We have presented a novel methodology to create a correct golden model in SystemC from a given textual specification. This initial modeling step has received little attention in the literature so far. In our methodology the two components of the specification – requirements and scenarios – are formalized to a set of properties and a testbench, respectively. The formalization step also produces a SystemC code skeleton, that is incrementally extended to a full ESL model in a verification-driven development process.

The proposed methodology has been demonstrated and shown to be promising for a case study. Clearly, this paper constitutes a first step towards a full verification-driven ESL design flow. Key challenges for future research are to improve the accessibility and the usability of the proposed general framework by automating its individual steps. This includes amongst other things automated assistance in the formalization process by using natural language processing techniques, and automatic generation of the code skeleton and also parts of the implementation. For the latter task, current methods such as [24], [25] can be adapted. For the former task, there also exist first approaches in related contexts. In [28] for instance an approach to generate an executable test environment from textual requirement specifications via UML class diagrams and the application of the classification tree methodology has been proposed. For automation of behavior driven development the integration of natural language processing has been considered in [29]. For future work, we also plan to use the proposed methodology for the development of a complex system.

References

[1] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann/Elsevier, 2007.
[2] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2011.
[3] Accellera Systems Initiative, "SystemC," 2012, available at http://www.systemc.org.
[4] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.
[5] D. Tabakov, M. Vardi, G. Kamhi, and E. Singerman, "A temporal language for SystemC," in *FMCAD*, 2008, pp. 1–9.
[6] *Accellera Property Specification Language Reference Manual, version 1.1*, http://www.pslsugar.org, 2005.
[7] S. Kundu, M. Ganai, and R. Gupta, "Partial order reduction for scalable testing of SystemC TLM designs," in *DAC*, 2008, pp. 936–941.
[8] L. Ferro and L. Pierre, "ISIS: Runtime verification of TLM platforms," in *FDL*, 2009, pp. 1–6.
[9] D. Tabakov and M. Vardi, "Monitoring temporal SystemC properties," in *MEMOCODE*, 2010, pp. 123 –132.
[10] A. Sen, "Concurrency-oriented verification and coverage of system-level designs," *ACM Trans. on Design Automation of Electronic Systems*, vol. 16, pp. 37:1–37:25, October 2011.
[11] D. Große, H. M. Le, and R. Drechsler, "Proving transaction and system-level properties of untimed SystemC TLM designs," in *MEMOCODE*, 2010, pp. 113–122.
[12] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri, "KRATOS: a software model checker for SystemC," in *Computer Aided Verification*, 2011, pp. 310–316.
[13] P. Herber, M. Pockrandt, and S. Glesner, "Transforming systemc transaction level models into uppaal timed automata," in *MEMOCODE*, 2011, pp. 161 –170.
[14] F. Rogin, R. Drechsler, and S. Rülke, "Automatic debugging of system-on-a-chip designs," in *IEEE International SOC Conference*, 2009, pp. 333–336.
[15] H. M. Le, D. Große, and R. Drechsler, "Automatic TLM fault localization for SystemC," *IEEE Trans. on CAD*, vol. 31, no. 8, pp. 1249–1262, 2012.
[16] H. M. Le, D. Große, and R. Drechsler, "Towards analyzing functional coverage in SystemC TLM property checking," in *HLDVT*, 2010, pp. 67–74.
[17] C. Kuznik and W. Mueller, "Functional Coverage-driven Verification with SystemC on Multiple Level of Abstraction," in *DVCON*, 2011, pp. 1–7.
[18] N. Bombieri, F. Fummi, and G. Pravadelli, "A mutation model for the SystemC TLM 2.0 communication interfaces," in *DATE*, 2008, pp. 396–401.
[19] P. Lisherness and K.-T. (Tim) Cheng, "SCEMIT: A SystemC error and mutation injection tool," in *DAC*, June 2010, pp. 228 –233.
[20] K. Beck, *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
[21] R. Mitchell, J. McKim, and B. Meyer, *Design by contract, by example*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2002.
[22] J. Östroff, D. Makalsky, and R. Paige, "Agile specification-driven development," in *Extreme Programming and Agile Processes in Software Engineering*, 2004, vol. 3092, pp. 104–112.
[23] H. Baumeister, A. Knapp, and M. Wirsing, "Property-driven development," in *SEFM*, sept. 2004, pp. 96 – 102.
[24] S. Uchitel, G. Brunet, and M. Chechik, "Synthesis of partial behavior models from properties and scenarios," *IEEE Trans. Software Eng.*, vol. 35, no. 3, pp. 384–406, 2009.
[25] I. Krka, Y. Brun, G. Edwards, and N. Medvidovic, "Synthesizing partial component-level behavior models from system specifications," in *ESEC/SIGSOFT FSE*, 2009, pp. 305–314.
[26] J. Li, F. Xie, T. Ball, V. Levin, and C. McGarvey, "Formalizing hardware/software interface specifications," in *ASE*, 2011, pp. 143–152.
[27] PROSYD, 2007, http://www.prosyd.org.
[28] W. Müller, A. Bol, A. Krupp, and O. Lundkvist, "Generation of executable testbenches from natural language requirement specifications for embedded real-time systems," in *DIPES/BICC*, 2010, pp. 78–89.
[29] M. Soeken, R. Wille, and R. Drechsler, "Assisted behavior driven development using natural language processing," in *TOOLS*, 2012, pp. 269–287.