LOCALIZING FEATURES OF ESL MODELS FOR DESIGN UNDERSTANDING

Marc Michael[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
{mmichael,grosse,drechsle}@informatik.uni-bremen.de

ABSTRACT

The increasing variety of functionality in embedded systems leads to more and more complex designs. Even abstraction techniques as applied in ESL design solve this problem only to a certain extend. In this paper we present an approach to improve design understanding of ESL models. Our approach localizes features by comparing simulations on ESL models. Code coverage techniques are used to highlight source code implementing a certain feature. This significantly helps the different team members since for major design tasks like e.g. refinement, partitioning or optimization it is required to know where a certain functionality has been implemented. We demonstrate the advantages of our approach for a complex image processing system.

1. INTRODUCTION

Each new generation of embedded systems adds a huge amount of new functionality. This steady progress is only possible by using abstract modeling. Hence, an ecosystem around *Electronic System Level* (ESL) design has been developed by the EDA companies. A major language for ESL design is the C++ class library SystemC [1, 2, 3, 4]. Besides abstract modeling, SystemC supports hardware/software co-design and the creation of heterogeneous systems (containing digital, analog and RF). A major approach in ESL design is the creation of advanced virtual prototypes. These prototypes heavily use *Transaction Level Modeling* (TLM) to abstract functionality, communication and timing [5]. Based on the SystemC TLM 2.0 standard [6] TLM models can be easily re-used in different projects and IP integration has become an easier task. Overall, high-speed simulation, architecture evaluation and early software development is standard practice in SystemC-based ESL flows today.

However, the steadily increasing functionality results in large and complex ESL models, even if unimportant details at the higher levels have been abstracted using TLM. Thus, understanding these models is a non-trivial task. During the design of an ESL model the specification is implemented as a large set of system features. Following the IEEE Standard 829 [7] a *feature* is a distinguishing characteristic of the ESL model. More specifically, besides concrete functional features (e.g. running a video filter, converting audio data) also other features such as performance or dependability are important properties of an ESL model.

In this paper, we propose an approach to improve design understanding of ESL models by localizing features automatically, that is, identifying the respective source code implementing the concrete feature. The approach is based on simulation and works as follows: We compare a simulation run where the feature is activated against a run where the feature is not active. During simulation, code coverage techniques are used such that we can determine the differences between both runs. Essentially, this delta identifies the relevant source code implementing the feature.

Major ESL design tasks like for instance refinement, partitioning or optimization heavily depend on this information. In general, since the implementations of different features is typically spread over several team members this knowledge is distributed. Furthermore, the integration of new members (e.g. system architects, software developers or hardware engineers) is accelerated since the design familiarization based on our approach not only consists of reading long text books and manual code inspection.

In summary, the contributions of this paper are:

- Pinpointing to relevant source code implementing a feature
- Following the information flow when presenting the feature code
- Highlighting involved SystemC modules

In an experimental evaluation we demonstrate the advantages of our approach for a complex image processing system. Different features can be automatically localized very fast with very little user interaction. In this way, design understanding is significantly improved.

The remainder of the paper is structured as follows: Section 2 describes related work. In Section 3 our feature localization approach for SystemC ESL models is introduced. The evaluation is given in Section 4. Finally, the paper is concluded and future work is discussed in Section 5.

2. RELATED WORK

So far no feature localization approach for ESL models has been presented. However, feature localization has been studied intensively in the context of software engineering. The goal of [8] is to help the software engineer during the process of software maintenance, i.e. if the functionality of a software system needs to be updated or extended. Different test-cases are executed and in combination with a test coverage monitor the approach can be used to discover where a particular program feature is implemented. A quite similar approach has been presented in [9]. Both approaches consider a set of test-cases and compute a categorization. These dynamic approaches have been advanced by incooperating static analysis techniques. In [10] computational units are identified that specifically implement a feature as well as the set of jointly or distinctly required computational units for a set of features.

In the context of fault localization, coverage-based methods have been developed to represent how source code lines act during passed and failed tests [11]. Several coverage metrics can be integrated to improve the results [12].

A method for feature localization of hardware designs has been presented in [13]. The method uses line coverage and toggle coverage to find relevant HDL code for a feature. However, RTL designs are addressed not ESL models.

3. FEATURE LOCALIZATION FOR ESL MODELS

In this section the proposed approach for feature localization in SystemC ESL models is introduced. First, the general idea is described. Then, we present our method in more detail.

3.1. General Idea

For finding a feature in a SystemC ESL model two simulation scenarios are required. The first scenario needs to include the requested feature and the second scenario must not include the feature. Then, by comparing the activated source code of both scenarios after simulation we can extract the source code which has only been executed in the first scenario. These code lines show the implementation of the requested feature. The involved SystemC modules can now be highlighted. If more than one SystemC module is involved, the respective source code is presented with regard to the activation during simulation.

In the following we describe our approach in more detail. At first, we explain the terms feature and scenario for SystemC ESL models. Then, we define the problem and present our implementation.

3.2. Features and Scenarios

Several definitions of a feature have been proposed in the software community, in particular in the context of feature-oriented software development (for an overview see [14]). Since we deal with abstract SystemC models, a *feature* is a unit of behavior that satisfies a functional and/or non-functional requirement. For example, a feature could be to open the menu, to save a picture, or to save the current state of a system. This feature definition is clearly motivated by the high level of abstraction at ESL.

A *scenario* is a sequence of inputs to a SystemC TLM model. Such a scenario activates a feature if the result of the feature can be observed when executing the scenario on the SystemC TLM model.

Here is a concrete scenario example of taking a picture using a camera:

- Pressing *power* to turn on the system
- Activating the camera by pressing *camera mode*
- Taking a picture by pressing the *shutter release*
- Showing the taken picture by pressing *play mode*
- Grayscale the image by selecting the *grayscale operator*
- Turning off the system by pressing *power*

Based on these terms, we give the problem formulation in the next section.

3.3. Problem Formulation

Before we get into more detail, we provide the used notations:

- S denotes the set of scenarios, and
- $F = \{F_1, ..., F_m\}$, $m \in \mathbb{N}$ denotes the set of features.

For localization of the implementation of the feature F_i we need two scenarios, i.e. S_a and S_n, respectively. S_a and S_n are almost identical except that S_a activates F_i and S_n does not activate F_i. Now, the desired source code implementing F_i can be determined using code coverage techniques as:

$$cov(S_a) \setminus cov(S_n) = code(F_i),$$

where $cov(A)$ returns the covered source code lines of scenario A, and $code(B)$ represents the relevant source code lines of feature B.

In the next section an example and the implementation of this approach is introduced.

3.4. Example and Implementation

As mentioned in the previous section two scenarios activating (S_a) and not activating (S_n) the feature need to be defined by the user. This is a manual step. For the creation of these scenarios the specification, documentation or existing test cases can be used.

Assume we have an ESL model for a digital camera and we want to localize the feature F_1: grayscaling an image. As activating scenario S_a we use the scenario from Section 3.2. For the non-activating scenario S_n we use S_a but

remove pressing the grayscale operator. Now the SystemC ESL model is simulated with scenario S_a and afterwards with scenario S_n. During both simulation runs we use gcov [15], a test coverage program for C++, to log how often each line of code executes and what lines of code are actually executed. A concrete example giving partially logged coverage information is:

```
        -:  174: // create grey image
     3456:  175: for(y=0;y<height;y++)
        -:  176: {
 17915904:  177:   for(x=0;x<width;x+=3)
        -:  178:   {
 17915904:  179:     red    = *(image+(y*width)+x);
 17915904:  180:     green  = *(image+(y*width)+x+1);
 17915904:  181:     blue   = *(image+(y*width)+x+2);
 17915904:  182:     *(grey+(y*grey_width)+(x/3))
                       =   0.299*red
                         + 0.587*green
                         + 0.114*blue;
        -:  183:   }
        -:  184: }
```

The first number in each row shows how often a line of code has been executed. The second number is the actual line number followed by the original source code line. In the concrete example an $5,184 \times 3,456$ image has been used.

By comparing the logged coverage information of S_a and S_n we can extract the source code lines which have only been executed in one scenario, i.e. in S_a. Hence, these lines of code contain the implementation of the requested feature F_i (see above for parts of this code for feature F_1). Moreover, after localizing the relevant code lines the involved SystemC modules are known, too. By using visualization techniques similar to [16, 17] we can additionally provide a graphical representation. In general, the modules and their interconnection is available. But with the localized source code we can now only highlight the modules which belong to the requested feature. This information gives additional information on how the system is implemented and which module is responsible for which task.

If the source code of a feature is spread over several SystemC modules all involved modules are highlighted. For a better understanding of how the feature is implemented all involved modules are ordered with respect to their execution. In that way we can visualize where a feature starts and how the execution continues. To get the correct order of the involved modules a second run of the scenario which includes the feature is required. On the second run time-stamps are added to the determined lines of code which enables to compare the ordering.

In the following section we present the experimental evaluation.

4. EXPERIMENTAL EVALUATION

This section presents the experimental results for the proposed approach. At first, we describe our test environment. Then, we explain the test cases for our feature localization approach. Finally, the results are discussed.

4.1. Test Environment

We are using an improved version of the image processing system described in [18]. This system determines the position of a PlayStation™ Move Controller [19] in a video stream. On the top of the controller is an illuminated ball. We calculate its 3D position, i.e. x, y, z coordinates where the z coordinate can be derived from the radius of the ball.

Fig. 1 shows the architecture of the SystemC TLM model. The system includes the *Instruction Set Simulator* (ISS) Or1ksim [20] where the control software runs on.

The image capturing module receives the pictures from a camera and sends them via the bus to the memory. This module also initializes the image processing of the *grey* converter module which converts the colored image to a grayscale image. Then, edge detection is carried out in the *sobel* module. The result is used for performing a *Hough* transformation [21]. Now, the circle around the ball of the controller is determined (see *find circles* module). The position of the controller is then displayed on the video stream produced by the *image viewer*. For a more detailed description of the system we refer to [18].

The calculated position of the controller can be used to interact with the system. For example, starting a game which can be played via the controller. When starting the game a white circle, as depicted in Fig. 2, appears on the video stream. The goal of the game is to cover this circle with the top of the controller. If the circle is covered the player gets one point and the circle will appear at a new position. By reaching a certain number of points the level of difficulty will be increased by flipping and rotating the video which is computed in the *modifier* module.

For the evaluation process of our feature localization approach we replaced the camera stream with video files. Hence, we ensure that the input to the system is deterministic.

4.2. Test Cases

In Test Case 1 we want to add a new feature to the system. The system must be modified so that it is able to calculate the position of two controllers instead of only one. Our goal is to find feature F_1: calculating the position of the controller. To find F_1 we compare scenario S_{1a} where one controller is displayed on the video stream with scenario S_{1n} where no controller is displayed.

In Test Case 2 we aim to localize the feature F_2: switching to the next level. For localization we use the following scenario S_{2a} which activates F_2:

- Turning on the system by starting the simulation
- Starting the game with the controller
- Collecting points until we reach the second level

We compare S_{2a} with the almost identical scenario S_{2n}. The only difference to scenario S_{2a} is to get one point less than needed to reach the next level.

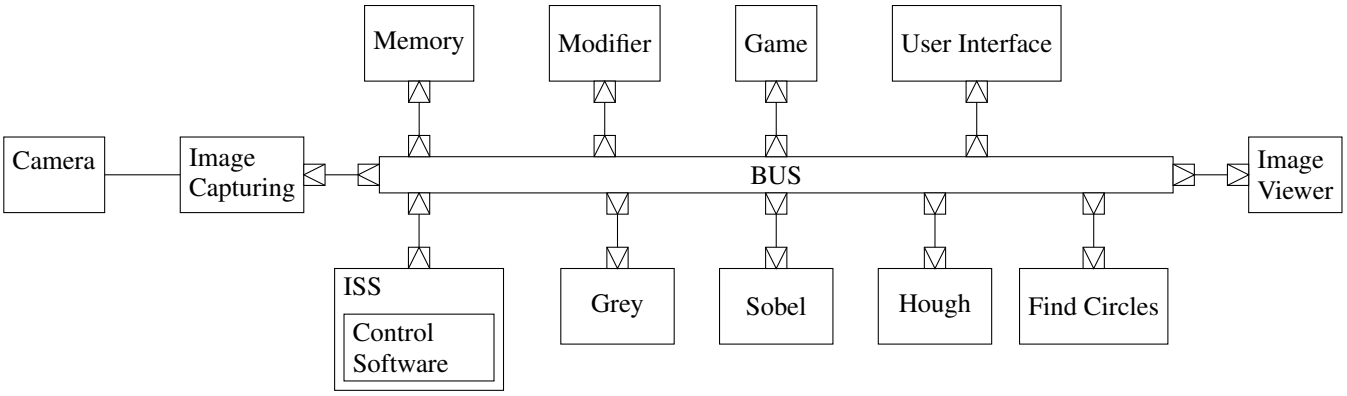

Fig. 1. Architecture of image processing system

Fig. 2. Example video output

To make the system robust different dependability measures have been implemented [18]. Hence, a damaged camera can be simulated to validate one of the dependability measures. The idea is to introduce some image artifacts using mutation on the image data. The respective mutations are injected using XML-definitions (e.g. size of the artifacts) following [18]. Now, in Test Case 3 the feature F_3 is: controlled modification of image data. For our approach we use a standard scenario but include the injection of 5% random artifacts for S_{3a}, and only the standard scenario without any error injection as S_{3n}, respectively.

Test Case 4 also simulates an error similar to Test Case 3. However, Test Case 4 represents a corrupted connectivity between the *bus* and the *grey* module. The corrupted connectivity is represented as mutated TLM attributes, in particular the TLM command and the data length. Unlike Test Case 3 it is more important if the mutations can be fixed and where they will be fixed. The goal is to locate feature F_4: error correction of communication. Like in S_{3a} of Test Case 3 we activate in S_{4a} the communication errors, and in S_{4n} no errors are injected.

Table 1. Experimental results

Feature	# Modules	# Methods	# LOC
F_1	2	2	35
F_2	2	2	19
F_3	0	3	141
F_4	1	5	192

4.3. Results

Table 1 summarizes the results of all test cases. The first column *Feature* gives the requested features of the test cases. Column *Modules* gives the number of modules which include located source code of the feature. The respective number of methods are given in column *Methods*. Finally, column *LOC* reports the actual number of lines of code for the identified relevant source code implementing the feature.

In Test Case 1 two modules are involved. In each module one method is invoked. The automatically created graph of the system (see Fig. 3) shows that the module *find circles* has been executed before the module *image viewer*. The identified method of the first module *find circles* directly points to the code we are interested in: With the Hough transformation all circles on the current image are computed and the final result is the one with the best fitting circle with respect to the dimension of the PlayStation controller. The module *image viewer* draws only the position of the controller if it has been found.

In Test Case 2 two relevant SystemC modules have been identified by our approach. Fig. 4 shows the involved modules for feature F_2 in the order of execution, i.e. at first the *game* module is active followed by the *modifier* module. A screen shot showing the relevant feature code is depicted in Fig. 5. As can be seen, variable mem−>level of module *game* will be increased when points%points2nextLvl becomes zero. As soon as mem−>level will be increased from 0 to 1 the code in the SystemC *modifier* module becomes active (see Fig. 6). In this case the input image from the camera will be swapped vertically. Using this information a developer can easily add

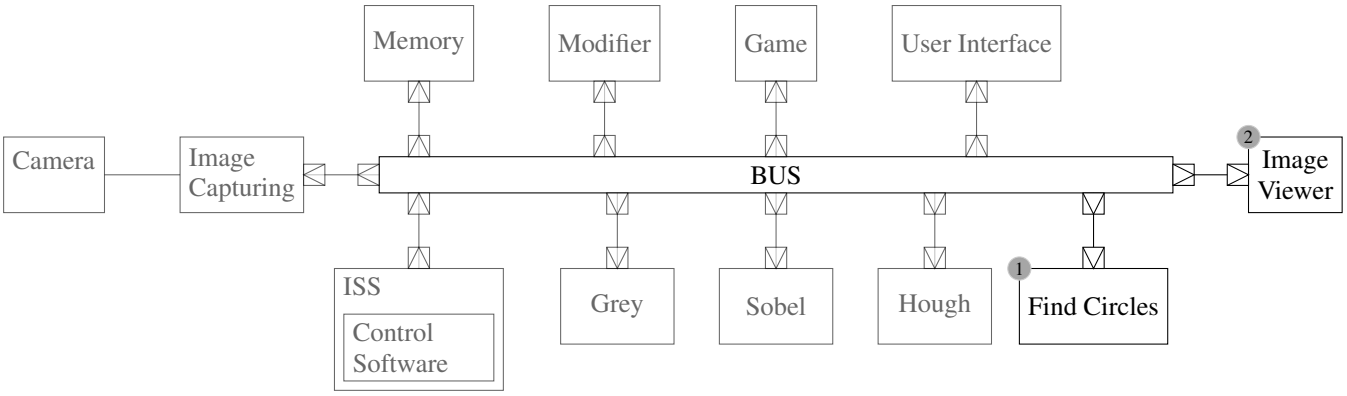

Fig. 3. Involved modules for Test Case 1

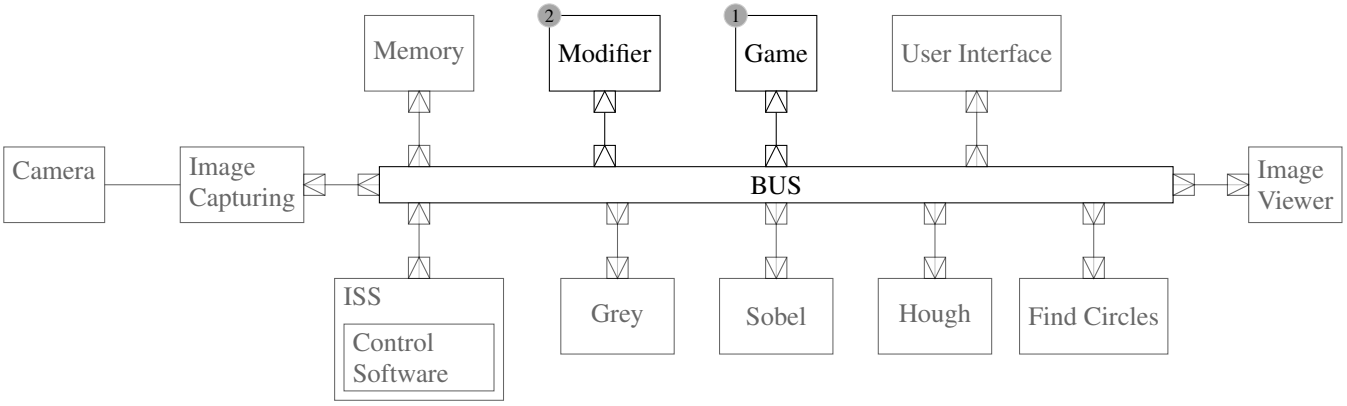

Fig. 4. Involved modules for Test Case 2

additional conditions for reaching the next level or changing the axis for swapping the image.

In Test Case 3 no modules of the system are involved for feature F_3. The modification has been implemented in the extended version of the TLM library (see [18]). Three methods have been identified to be relevant by our feature localization approach. Taking a closer look on these methods we saw that one method is responsible to load the specified modifications from the XML file, the second method is looking for modifications and calls the third method if modifications are specified.

For Test Case 4 our approach also localized code of the extended TLM library to be relevant for feature F_4. Moreover, two methods in the *grey* module are located which correct the communication error.

In summary, our feature localization approach clearly improves design understanding since automatically relevant source code as well as involved modules (and methods) for a feature are identified.

5. DISCUSSION AND FUTURE WORK

In this paper we have presented an approach for feature localization in ESL models. Basically, we compare a run which invokes the feature to a run which does not invoke the feature. Based on code coverage techniques we then compute the relevant source code implementing the feature. Besides the code we are also able to show the involved SystemC modules according to their order of activation. For a complex SystemC ESL design, implementing a video image processing system, we localized different features very fast.

In future work we want to increase the accuracy of our approach and to support situations where for example two features run in parallel. Therefore we plan to consider more than two scenarios as well as structural information. This allows for example to identify shared code of several features as well as more exact code localization.

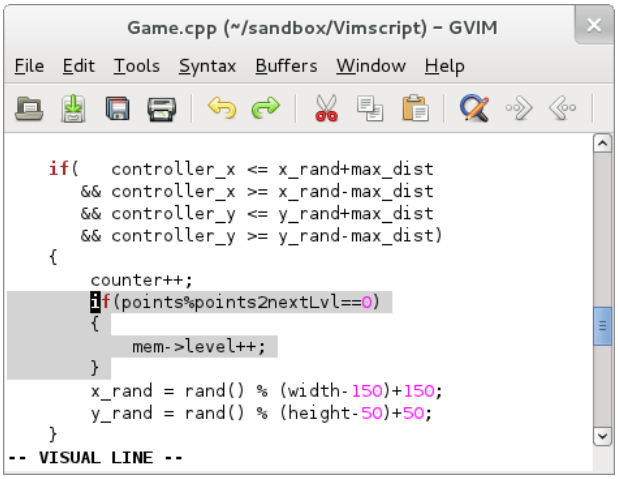

Fig. 5. Highlighted code in game module of feature F_2

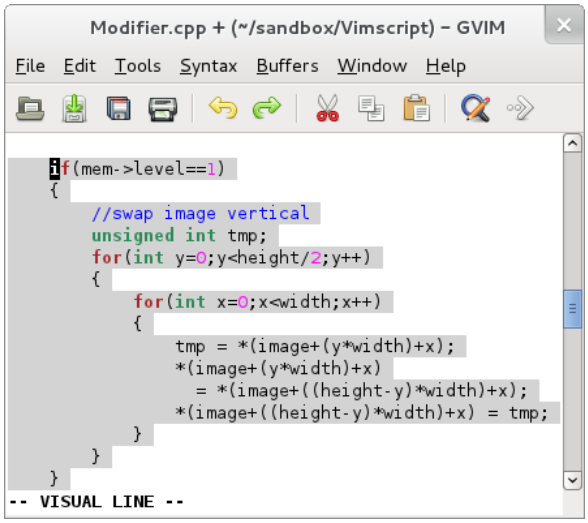

Fig. 6. Highlighted code in modifier module of feature F_2

6. ACKNOWLEDGEMENTS

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.

7. REFERENCES

[1] Accellera Systems Initiative, "SystemC," 2012, available at http://www.systemc.org.

[2] D. C. Black and J. Donovan, *SystemC: From the Ground Up*. Springer-Verlag New York, Inc., 2005.

[3] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.

[4] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2005.

[5] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2003, pp. 19–24.

[6] J. Aynsley, *OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL*. Open SystemC Initiative (OSCI), 2009.

[7] "IEEE standard for software and system test documentation," *IEEE Std 829-2008*, 2008.

[8] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," *Journal of Software Maintenance*, vol. 7, pp. 49–62, January 1995.

[9] W. Wong, S. Gokhale, J. Horgan, and K. Trivedi, "Locating program features using execution slices," in *Application-Specific Systems and Software Engineering and Technology, 1999. ASSET '99. Proceedings. 1999 IEEE Symposium on*, 1999, pp. 194 –203.

[10] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Trans. Software Eng.*, vol. 29, no. 3, pp. 210–224, 2003.

[11] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization for fault localization," in *Proceedings of the Workshop on Software Visualization*, 2001.

[12] R. A. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *ICSE*, 2009, pp. 56–66.

[13] J. Malburg, A. Finder, and G. Fey, "Automated feature localization for hardware designs using coverage metrics," in *Design Automation Conf.*, 2012, pp. 941–946.

[14] S. Apel and C. Kästner, "An overview of feature-oriented software development," *Journal of Object Technology (JOT)*, vol. 8, no. 5, pp. 49–84, July/August 2009.

[15] "Gcov - using the gnu compiler collection (gcc)," http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[16] D. Große, R. Drechsler, L. Linhard, and G. Angst, "Efficient automatic visualization of SystemC designs," in *Forum on specification and Design Languages*, 2003, pp. 646–657.

[17] C. Genz, R. Drechsler, G. Angst, and L. Linhard, "Visualization of SystemC designs," in *IEEE International Symposium on Circuits and Systems*, 2007, pp. 413–416.

[18] M. Michael, D. Große, and R. Drechsler, "Analyzing dependability measures at the Electronic System Level," in *Forum on specification and Design Languages*, 2011, pp. 1–8.

[19] "Playstation™ move motion controller," http://uk.playstation.com/psmove.

[20] J. Bennett, *Or1ksim User Guide*, 2010.

[21] D. Ballard, "Generalizing the hough transform to detect arbitrary shapes," *Pattern Recognition*, vol. 13, no. 2, pp. 111 – 122, 1981.