Automated Feature Localization for Dynamically Generated SystemC Designs

Jannis Stoppe[1] Robert Wille[1,2] Rolf Drechsler[1,2]
[1]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[2]Group of Computer Architecture, University of Bremen, 28359 Bremen, Germany

***Abstract*—Due to the large complexity of today's circuits and systems, all components e.g. in a *System on Chip* (SoC) cannot be designed from scratch anymore. As a consequence, designers frequently work on components which they did not create themselves and, hence, design understanding becomes a crucial issue. Approaches for feature localization help by pinpointing to distinguished characteristics of a design. However, existing approaches for feature localization mainly focused on the Register Transfer Level; existing solutions for the Electronic System Level (using languages such as SystemC) have severe limits. In this work, we propose an approach for advanced feature localization in SystemC designs. By this, we overcome major limitations of previously proposed solutions, in particular the missing support for dynamically generated designs, while keeping the proposed solution as non-intrusive as possible. The benefits of our approach are confirmed by means of a case study.**

I. Introduction

Current chip designs are becoming more and more complex. As designs tend to shift towards *System-on-Chips* (SoCs) and even *Networks-on-Chips* (NoCs), both, the amount of features realized in a single design and the amount of atomic elements needed to realize that functionality is growing significantly. As a consequence, such designs are increasingly realized through the re-use of existing parts and external *Intellectual Property* (IP) [1]. These parts may even form a hierarchy, with complex functionality being implemented in blocks that, in turn, are composed of several blocks themselves. This results in a complex functionality being implemented in various layers across the design. Additionally, more designers are usually collaborating to work on a single design. As designs get larger, a separation of concerns is usually carefully organized for both, the design and the people working on it. By this, designers are enabled to work in large teams on a single system [2]. Consequently, designers frequently need to work on parts of an implementation that they are not familiar with. Hence, establishing the needed *design understanding* as quickly as possible is crucial [3].

Methods for *feature localization* provide an alternative if a documentation is e.g. outdated or incomplete. They aid designers by pinpointing them to distinguished characteristics of a design and, by this, allow them to quickly locate implementations of certain features of a system. Supported by that, the designer avoids a manual inspection of large parts of the design and can directly focus on those parts that matter for the currently considered design task.

For this purpose, several methods for feature localization have been proposed (see e.g. [4], [5]). The underlying techniques usually involve running several simulations which are supposed to trigger certain features. At the same time, it is traced which parts of the design were used in the respective run. Based on that information, the implementation of a given feature can usually be located in the given implementation. But while these feature localization techniques provide an effective way to direct a designer to the part of the design that is relevant for the current task, most of the existing solutions can only be applied to designs at the *Register Transfer Level* (RTL).

However, in order to meet the demand of shorter development cycles and working prototypes early in the design process [6], systems are increasingly designed at the more abstract *Electronic System Level* (ESL) – which motivates the need for feature localization for this abstraction level. Unfortunately, corresponding support for implementations in SystemC – the current de-facto standard at the ESL [7] – is very limited thus far. To the best of our knowledge, the only approach for feature localization at the ESL has been presented in [8].

In this work, we propose an approach for advanced feature localization. For this purpose, a smart combination of SystemC/C++ utilities for line coverage analysis such as *gcov* together with the scheme of *Aspect-Oriented Programming* (AOP, [9]) is applied. By this, a hybrid static/dynamic coverage metric is introduced that enables the designer to precisely get pinpointed to features in existing SystemC designs. The proposed solution clearly overcomes the limitations of previously proposed approaches, i.e. additionally considers dynamically generated designs, while remaining as non-intrusive as possible and, hence, applicable to a wide variety of SystemC projects.

II. Feature Localization

Designers need to be able to understand the respective components of a design as well as their relation to each other. In particular, they need to efficiently grasp all the *features* of a system which are important to implement a particular improvement, extension, or bugfix. Those features are usually distinguished characteristics of a design, defining the expected output or behaviour of the system resulting from a specific input.

However, identifying those features and, by this, getting a sufficient understanding of the design in acceptable time is a cumbersome task. Often, the documentation is not as detailed as needed or became obsolete due to changes in the implementation that have not entirely been propagated [10]. As a consequence, *feature localization* is a crucial step within the design process which, thus far, has mainly been conducted manually (e.g. by inspecting the HDL implementation). This added a time-consuming and, hence, cost-intensive step to today's design flows in which neither any new functionality is added nor a single bug is fixed.

In order to aid this process, researchers developed automatic methods for feature localization (see e.g. [4], [5]). They aid the designer by automatically locating features based on so-called *Coverage Items* (CIs), i.e. parts of the implementation whose execution can be tracked. If the execution of these CIs is tracked, a designer can easily check whether a particular feature does or does not depend on the respective parts of the implementation. More precisely, if an execution (or a *run*) triggering a CI includes the feature the designer is looking for, he/she can conclude that the respective parts of the implementations may relate to the considered feature. Performing several runs, possible CIs and, by this, responsible parts of the implementation can further be refined.

Overall, methods for feature localization narrow down the items, i.e. the parts of the implementation, that need further inspection to a tiny fraction. By this, they significantly aid

978-3-9815370-4-8/DATE15/©2015 EDAA

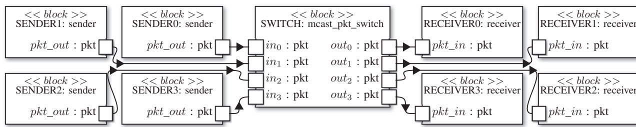

Fig. 1. Simplified representation of an initialized *pkt_switch* system.

designers to quickly determine the relevant parts of the implementation in order to conduct their improvements, extensions, or bugfixes. However, existing solutions for feature localization have a severe limitation which is discussed next.

In fact, feature localization of SoCs mainly focused on the *Register Transfer Level* (RTL) and its corresponding programming languages such as VHDL [4], [5]. However, due to the increasing complexity, designers strive for higher level of abstractions – particularly in early stages of the design process. In fact, design at the *Electronic System Level* (ESL) with SystemC as its de-facto standard [7] is established in industry today. But here, automatic feature localization is limited significantly – particularly due to the missing support of dynamically generated designs in SystemC.

In fact, to the best of our knowledge, the only approach for feature localization at the ESL has been proposed in [8]. Here, motivated by the fact that SystemC is purely based on C++ [11], existing C++ coverage tools such as *gcov* and the respective coverage metrics have been applied. But this approach is restricted to the static code description of the given SoC. The dynamic behavior supported by SystemC – particularly differentiating between multiple instantiations of the same type of class – are not supported. As a consequence, features might not be trackable.

Example 1. *Consider the simplified representation of the* pkt_switch *system, one of the standard examples which are provided by SystemC, as shown in Fig. 1. The switch in the center distributes data and is connected to four senders and receivers which generate and receive arbitrary packages, respectively. The senders (receivers) are instances of a respective class* `sender` *(*`receiver`*) and, therefore, rely on the same source code.*

This leads to severe problems for automatic feature localization which entirely relies on a static view of the source code. In fact, those methods are unable to differentiate between a feature that is statically defined in the source code and a feature that is dynamically defined through the instantiation within the elaboration phase. As an example, those approaches would not be able to differentiate between the feature "send to 0" and "send to 1".

III. Proposed Solution

In this section, an approach for an advanced feature localization for SystemC designs is proposed. The main goal is to overcome the limitations of previously proposed solutions, i.e. the static focus and, hence, missing support for dynamically generated designs. At the same time, we aim for keeping the proposed solution as non-intrusive as possible, i.e. we propose a solution which prevents the designer from significant changes of their implementation just for feature localization. In order to accomplish that, a smart combination of SystemC-utilities for line coverage analysis such as *gcov* together with the scheme of *Aspect-Oriented Programming* (AOP, [9]) is applied. The next section introduces the general idea of the presented approach. Afterwards, specifics concerning AOP and its application for SystemC are described. The benefits of the solution are later outlined in a case study in Section IV.

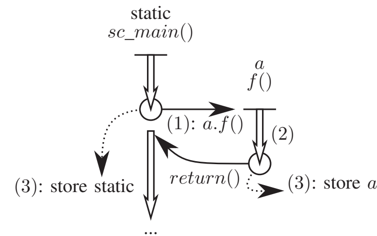

Fig. 2. The proposed coverage analysis scheme.

A. Considering Dynamically Generated Designs

Methods for the analysis of code coverage form the foundation for almost all approaches for feature localization [12], [13]. Line coverage e.g. provides a detailed per-line-of-source listing of parts of the program which have been executed in a given run. However, simply applying corresponding methods for line coverage analysis such as *gcov* does not always provide a satisfactory result. As discussed above, corresponding analyses are static and, while indeed pinpointing to respective lines of code, the precise instantiation (and by this the precise component of the design dynamically generated in the elaboration phase) would remain unknown.

In order to address this, we propose the insertion of another dimension when performing line coverage analysis. Instead of only providing the designer with the total amount of times a single line has been executed during a run, we aim for providing the designer with the total amount of executions of a line *per instance*.

Example 2. *Let's assume a system in which two components a and b are dynamically generated from a common base code. Furthermore, let's assume that, during simulation, only component a is triggered. A static analysis would only unveil that the respective lines of the common code basis have been triggered n times. In contrast, having the total amount of executions per instance would unveil that component a has been triggered n times while component b has not been triggered at all.*

Such a more elaborated result would lead to a much more accurate feature localization. In the example, it gets obvious that component a triggered the coverage item (and might include the feature the designer is looking for) while component b seems irrelevant in this case.

In order to differentiate which instance actually executes the code, the existing scheme is enriched by an analysis of the `this`-reference of the respective module. During the compilation of C++ programs, functions that are not to being executed from a static context get an additional parameter; the `this`-pointer. This reference allows access to the instance of the module which actually executes the respective function. Using the `this`-pointer, one can track a unique identifier – e.g. the `name`-field of the respective SystemC-object – and, by this, keep track of which instance is currently executing a particular part of code.

Relying on these ideas, feature localization is improved as sketched in Fig. 2: Whenever a function is called (1), a common code line analysis tool (e.g. *gcov*) is additionally invoked. This keeps track of which lines are triggered when running the function (2) – as in any existing feature localization tools. However, the tracked information is not stored globally, but with respect to a unique identifier of the instance which runs the function. For this purpose, the execution of the analyzer is terminated together with the currently considered function (3). In order to determine the unique identifier, the `this`-pointer is read just before the execution of the function is terminated. Invoking a classic coverage tool at each of those points and

mapping the information from this tool to the currently active `this`-pointer records the full line-based coverage and adds the information which object instance the according lines were called from.

By this, dynamically generated components are explicitly considered for feature localization. However, a naive realization of this solution is not practical. In fact, a straightforward implementation of the proposed scheme would require designers to perform significant changes in their existing project. For each function, the respective parts for step (1) and step (3) from Fig. 2 would have to be added – something which should be avoided as it would introduce a lot of modifications to the code that do not add any functionality and take a considerable amount of time to write. Hence, in order to realize the proposed scheme in a non-intrusive fashion, one further measure is applied: the exploitation of aspect-oriented programming.

B. Exploitation of Aspect-Oriented Programming

AOP is a programming scheme motivated by the following scenario frequently occurring in system design: A new functionality shall be implemented in an existing system which, however, would require an extensive re-factoring of the existing implementation (or even an entire re-development from scratch).

Example 3. *Consider again the `pkt_switch`-system from Fig. 1. Each of the classes used in the implementation contains an `entry()`-method that updates the state and output of the respective module. Let's assume that this design shall be enriched with a tracing functionality which keeps track of each call of the respective `entry()`-methods. Although all modules of that example share this method, they do not inherit it from a common base class. Hence, there is no single location to add the respective extensions to. The designer is left with the option of either*

- *enriching the entire system e.g. by an inheritance structure (which might be a lot of work and may result in other designers having to adapt their code as well) or*
- *adding the respective code in all classes (which results in redundant structures or global methods that are supposed to be called only from a certain context – both considered bad style which decreases maintainability).*

For those cases, AOP provides the designer with an additional layer that allows him/her to describe the new behaviour (almost) independently from the existing implementation. Following this scheme, designers avoid huge re-factorings but need to provide the implementation of the newly added behaviour *and* a description of the position it is supposed to be executed at. More precisely, AOP distinguishes between separate *component code* (which represents the existing object-oriented programming scheme that is used to describe a certain structure and basic functionality) and a so-called *aspect code* (which describes additional functionality that may be shared by several components and does not fit into the existing structure). These two kinds of code are written independently and, before compilation, are merged by the so-called *aspect weaver*. The aspect weaver takes the aspect code and inserts it into the specified positions (so-called *join points*) in the original source code.

Example 4. *Using AOP, the desired tracing functionality from Example 3 can be realized by leaving the existing code as it is. Instead, the tracing functionality is separately realized in an aspect. This aspect code also specifies that all classes inheriting the `sc_module`-class and containing an `entry()`-method (i.e. sender, receiver, and switch) should execute the newly added tracing implementation whenever `entry()` is called. The resulting code consists of the original SystemC model as well as a description of additional functionality that describes where in the design (i.e. "after calling the `entry()`-method") which functionality (i.e. "trace the execution") should be added.*

The ability to transparently "weave in" aspect code just before compilation allows for the insertion of additional functionality to large source bases without further interaction. Other designers of a given system do not even need to know about new features being added: functionality that is needed at some other point in the workflow can remain hidden from them. This results in less complexity as designers are only presented with implementations related to their respective tasks – a clear separation of concerns.

In order to apply AOP in SystemC, or C++ in general, *AspectC++* [9] provides a proper implementation of the respective programming scheme. AspectC++ performs thereby two steps:

- First, *weaving* is applied, i.e. the original source code as well as the respective aspect source code (which contains its own keywords to specify the aspects but largely sticks to the C++'s syntax) is taken and a corresponding woven source is created. This code includes both, the original functionality and the parts introduced in the aspects.
- Afterwards, the resulting source code (which is not necessarily meant to be read or edited) is *compiled*; directly leading to a binary executable which allows to perform the new functionality.

C. Integration and Discussion

Following the AOP-based programming scheme allows for a non-intrusive integration, i.e. the analysis functionality proposed in Section III-A can easily be integrated into existing SystemC projects. In fact, since SystemC is a C++ library, AspectC++ can be used with hardly any changes. Only the weaver needs to be built into existing pipelines. However, this is a one-time setup and works on all major platforms [14]. Afterwards, a set of aspects can be used to add the new tracing functionality to any given SystemC design.

This results in a system, in which function calls are modified: First, information recorded thus far (by the respective line coverage analysis) is flushed prior to each function call. By this, all results of the analysis are saved (in our implementation, they are directly written to disk) and associated with the current scope (i.e. the instance which calls the function). Then, calling function f is executed. Since the line coverage analysis is still running, new information is gathered. Before the function terminates, another flush is executed, i.e. the newly collected information is saved again (now associated to the instance which executed f).

The result is a list of coverage analysis files – all associated to the instances on which the respective code was executed. Overall, this leads to several advantages which make the proposed solution very applicable for feature localization in SystemC designs.

- The original source code does not need to be modified in any way. Existing SystemC projects can easily be analyzed by only modifying the compilation workflow to include the aspect weaving step.
- The existing compilation setup can be used for the resulting, woven code, as long as the given compiler offers the coverage mechanisms that the designer wants to apply.
- The SystemC library does not need to be modified in any way. Unlike approaches that analyze a running SystemC design by modifying e.g. the simulation kernel, the given approach works purely on the user-generated code base. This means that the approach can be applied to future versions of SystemC without any further changes and that

TABLE I
RESULTS OF THE CASE STUDY

Feature	Coverage Items			[8]	this
	File	Instance	Lines		
to 0	receiver.cpp	RECEIVER0	40 – 41; 43; 45 – 51	✗	✓
	switch.cpp	SWITCH	194 – 195	✓	✓
to 1	receiver.cpp	RECEIVER1	40 – 41; 43; 45 – 51	✗	✓
	switch.cpp	SWITCH	201 – 202	✓	✓
to 2	receiver.cpp	RECEIVER2	40 – 41; 43; 45 – 51	✗	✓
	switch.cpp	SWITCH	207 – 208	✓	✓
to 3	receiver.cpp	RECEIVER3	40 – 41; 43; 45 – 51	✗	✓
	switch.cpp	SWITCH	213 – 214	✓	✓

it can be combined with setups that already rely on a modified SystemC library.

- The proposed approach is flexible, i.e. various tools and/or schemes for line coverage analysis can be chosen. Since tracking the information obtained by these tools/schemes can be realized using aspects, e.g. the time-consuming *gcov* operations can easily be omitted if performance is crucial.
- The proposed approach is platform-independent. As long as a corresponding line coverage analysis tool is available (which is the case for all major platforms), the proposed solution can be implemented. This is possible, because AOP-implementations such as AspectC++ are available for all major platforms as well.

Besides that, the benefits of the proposed solution has also been demonstrated by means of a case study. Corresponding results are summarized next.

IV. CASE STUDY

The solution proposed above has been implemented and evaluated by several case studies. In this section, the results are representatively discussed and compared to the approach presented in [8] – to the best of our knowledge the only approach for feature localization of SystemC designs available thus far. As representative, the *pkt_ switch*-system – one of the standard examples in the SystemC-library which has already been considered before in Fig. 1/Example 1 – is considered.

The design realizes a system for distributing data packages and is assumed to be instantiated with four `sender` and four `receiver` instances – all of them connected to a central `mcast_pkt_switch`. The senders create packages including a random payload which is distributed by the switch (running on a slower clock) to the receivers. The system has already been illustrated before in Fig. 1.

The features a designer may look for in this system may be concerning the distribution of data. In particular, which parts of the design are triggered when a package is sent to a specific destination might be of interest. For this purpose, features *send to 0*, *send to 1*, *send to 2*, and *send to 3* are defined which are supposed to pin-point the designer to parts of the implementations where the delivery of packages to receiver 0, receiver 1, receiver 2, and receiver 3 are realized, respectively.

For the actual feature localization, five runs are performed. One run simulates the original setup where packages with an arbitrary payload are delivered to an arbitrary receiver. The other runs simulate the delivery to one specific receiver (as there are four receivers, four additional runs are required for this). We configured the feature localization so that only coverage items are reported which are *always* triggered when also the respectively feature was triggered and vice versa. This is sufficient for the considered purpose; however, alternative criteria as discussed e.g. in [15] could easily be considered as well (e.g. being *sometimes* triggered in runs that do not contain the considered feature).

For all considered features, Table I provides the result obtained by the approach previously proposed in [8] and obtained by the approach proposed in Section III. As it can clearly be seen, the previously proposed approach pin-points the designer only to parts of the code related to the switch-module. This can easily be explained by the fact that the dynamically instantiated receivers (where the feature that something is sent to them is realized) are not considered by the purely static approach from [8]. In contrast, the proposed methodology does not only pinpoint the designer to the switch but in addition to that even to the respective parts of the receiver. Moreover, even the precise instantiation of the receiver is provided (see column denoted by *Instance*). Obviously, this provides a much more comprehensive feature localization.

V. CONCLUSION

In this work, we proposed a method for feature localization at the Electronic System Level, i.e. based on SystemC designs. For this purpose, a combination of SystemC-utilities for line coverage analysis with the scheme of Aspect-Oriented Programming has been applied. This enables the explicit consideration of dynamically generated designs while, at the same time, remaining as non-intrusive as possible. In fact, the proposed solution can be integrated into a variety of existing SystemC projects since neither the project source code nor the SystemC kernel or the compiler need to be modified in any way. As confirmed by case studies, the proposed approach generates results of much better quality.

VI. ACKNOWLEDGEMENTS

Supported by the Federal Ministry of Education and Research (BMBF) under grant 01IW13001 (SPECifIC) and the German Research Foundation (DFG) within the Reinhart Koselleck project under grant no. DR 287/23-1.

REFERENCES

[1] G. Martin, "Design Methodologies for System Level IP," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 286–289, IEEE, 1998.
[2] L. Dibiaggio, "Design Complexity, Vertical Disintegration and Knowledge Organization in the Semiconductor Industry," *Industrial and Corporate Change*, vol. 16, no. 2, pp. 239–267, 2007.
[3] U. Kühne, D. Große, and R. Drechsler, "Property Analysis and Design Understanding," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1246–1249, European Design and Automation Association, 2009.
[4] J. Malburg, A. Finder, and G. Fey, "Automated Feature Localization for Hardware Designs Using Coverage Metrics," in *Annual Design Automation Conference (DAC)*, pp. 941–946, ACM, 2012.
[5] J. Malburg, A. Finder, and G. Fey, "Tuning Dynamic Data Flow Analysis to Support Design Understanding," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1179–1184, IEEE, 2013.
[6] T. Rissa, A. Donlin, and W. Luk, "Evaluation of SystemC Modelling of Reconfigurable Embedded Systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 253–258, IEEE Computer Society, 2005.
[7] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rosentiel, "Object-Oriented Modeling and Synthesis of SystemC Specifications," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 238–243, IEEE, 2004.
[8] M. Michael, D. Große, and R. Drechsler, "Localizing Features of ESL Models for Design Understanding," in *Forum on Specification and Design Languages (FDL)*, pp. 120–125, IEEE, 2012.
[9] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language," in *International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, pp. 53–60, Australian Computer Society, Inc., 2002.
[10] D. L. Parnas, "Software Aging," in *International Conference on Software Engineering (ICSE)*, pp. 279–287, IEEE, 1994.
[11] P. R. Panda, "SystemC – A Modeling Platform Supporting Multiple Design Abstractions," in *International Symposium on System Synthesis (ISSS)*, pp. 75–80, IEEE, 2001.
[12] N. Wilde and M. C. Scully, "Software Reconnaissance: Mapping Program Features to Code," *Journal of Software Maintenance: Research and Practice*, vol. 7, no. 1, pp. 49–62, 1995.
[13] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight Fault-Localization Using Multiple Coverage Types," in *International Conference on Software Engineering (ICSE)*, pp. 56–66, IEEE, 2009.
[14] O. Spinczyk, D. Lohmann, and M. Urban, "Advances in AOP with AspectC++," in *International Conference on Intelligent Software Methodologies, Tools, and Techniques*, pp. 33–53, IOS Press, 2005.
[15] T. Eisenbarth, R. Koschke, and D. Simon, "Locating Features in Source Code," *Transactions on Software Engineering (TSE)*, vol. 29, no. 3, pp. 210–224, 2003.