Automated Feature Localization for Hardware Designs using Coverage Metrics[*]

Jan Malburg Alexander Finder Görschwin Fey
University of Bremen
28359 Bremen, Germany
{malburg, final, fey}@informatik.uni-bremen.de

Abstract

Due to the increasing complexity modern System on Chip designs are developed by large design teams. In addition, existing design blocks are re-used such that the knowledge about these parts of the design entirely depends on the quality of the documentation. For a single designer it is almost impossible to have detailed knowledge about all blocks and their interaction.

We introduce a simulation-based automation technique to support design understanding. Based on use cases provided by the designer and on their coverage information, the proposed technique identifies parts of the source code that are relevant for a certain functional feature. In two case studies the technique is shown to be at least as exact as reading the documentation with two important advantages: the automated approach is faster and more precise than the existing documentation for the inspected designs.

1. Introduction

Modern chip designs, especially Systems on Chip, grow with respect to their transistor count as well as their supported features. Such chips are developed by large design teams consisting of hundreds of people [ITR09] and are far beyond the point where a single designer knows every detail about the design. Furthermore chips are assembled of design blocks from different sources. A design block is a part of a chip which provides a defined functionality. The functionality ranges from very complex, for example complete CPU-cores, to simple encoder and decoder blocks. Typically, complex design blocks are assembled from several less complex design blocks. A design block could be a new block developed especially for the new chip, or it might be a block already used in previous designs, or even a third-party block. All of those design blocks in a chip are responsible for one or more different features. Some of the features might be realized by combining the functionality of several different blocks.

A feature is a distinguishing characteristic of a design. A functional feature defines the expected output of the system under specific input. Other types of features are for example robustness, defining the amount of errors which can occur before the result is incorrect or performance, limiting the time until the design has to return the expected output. In the following only functional features are considered and simply called features.

[*]This work was supported in part by the German Research Foundation (DFG, grant no. FE 797/6-1)

For design improvement, design extension, and bug fixing a developer has to understand the design. Those tasks become even more important, since the amount of re-used design blocks is continuously increasing in future [ITR09]. In order to understand a design it is mandatory to know where in the design which feature is implemented. This does not only mean to know the block providing a feature, but where exactly in the block the feature is implemented. In general, it is unlikely that a developer has this knowledge. Purely manual inspection of the *Hardware Description Language* (HDL) code is a laborious, and therefore cost intensive task and the developer still might miss relevant parts of the implementation. Therefore, it is desirable to have tools which help the developer to find the relevant code for a feature and to understand the design.
In this paper we present a technique for locating parts of the HDL code which are relevant for a functional feature. This technique uses a dynamic approach relating coverage information gathered by simulation to features executed by use cases. Results are presented to the user by coloring the source code.

The contributions of this paper are:

- a feature localization technique for hardware designs,
- a unified notation to compare existing coloring schemes,
- a new coloring scheme,
- the use of toggle coverage for feature localization, and
- an evaluation of the approach and coloring schemes using two open source designs.

The remainder of this paper is organized as follows: Section 2 gives an overview of related work. Notations are defined in Section 3. In Section 4 we present our technique and different heuristics for feature localization. Section 5 describes the application of our prototype to two open source designs. Section 6 concludes the paper and discusses results.

2. Related Work

So far, no technique has been published about feature localization in hardware designs at the HDL level. However, feature localization for software designs is an area well-studied for over 15 years. Often the statement coverage from runs using a wanted feature is compared to the coverage of runs not using this feature [WS95]. Simple approaches only consider program statements which are covered by runs using a wanted feature, but are not covered by runs that do not use the feature [WC96]. More advanced approaches use more fine-grained categorizations, where the statements are classified based on the relation of runs using (not using) a certain feature [EKS03].
Techniques for bug localization in software using coverage information are similar to coverage-based feature localization. A well-known tool in this context is *Tarantula* [JHS01]. *Tarantula* is a visualization tool for bug localization which colors statements depending on their suspiciousness of causing a bug. The suspiciousness is computed by comparing the percentage of failing runs to non-failing runs executing the statement. Abreu et. al. [AZvG06] showed that using the *Ochiai coefficient*, a similarity coefficient used in molecular biology, yields better results for computing the suspiciousness of a statement compared to the *Tarantula* formula. Later Santelices et. al. [SJYH09] presented an approach to relate branch- and definition-use-coverage to statements. They showed that using the average of several different coverage criteria to compute the final suspiciousness creates better results than each coverage criterion on its own.

The approach presented in this paper is based on coverage information gathered by simulating the design under test. To this extend it is similar to the approaches described above. However, all previous approaches are used for software, while we consider hardware systems. There are several differences between software and hardware. For hardware there exist different coverage metrics, like toggle coverage, which we also use for feature localization. Moreover, in HDL descriptions of a design, there is several code which continuously is executed without being called from any other function, for example *always-blocks* and *assign-statements* in Verilog [IEE06]. Finally, hardware designs are inherently parallel.

3. Preliminaries

In this section we give basic definitions and introduce some terminology required for the rest of the paper. Let D be the design under test. A *use case* u for D is given by a sequence $u = (i_1, i_2, ..i_m)$ of input values i for D. A use case may either be directly defined by the user, or a test case from the test bench of D may be considered as use case. A feature f is a required behavior of D, defined by the developer or the specification of D. The set $F = \{f_1, f_2, f_3, .., f_k\}$ is the set of all features supported by D. A run r is the simulation of D applying a use case. The user defines, if a run r *uses* a feature f. The set $R = \{r_1, r_2, r_3, ..., r_n\}$ is the set of all runs. A *coverage metric* C with respect to D is a set of conditions. A *coverage item* $c \in C$ is a single condition. The form of these conditions is defined by C. Two typical coverage metrics used in hardware design are statement coverage and toggle coverage [TK01]. Toggle coverage is a coverage metric especially for hardware design, on the other hand statement coverage is also used in software design [JHS01]. In case of statement coverage C_s, for each statement s contained in the HDL code of D, there exists exactly one condition c_s, where c_s has the form "r executes s". In case of toggle coverage C_t for each wire and each register t there exist exactly two conditions c_{t_1} and c_{t_2}, where c_{t_1} is of the form "r switches t from 0 to 1" and c_{t_2} is of the form "r switches t from 1 to 0". The set $\mathcal{C} = C_s \cup C_t \cup ...$ is the union of all coverage metrics with respect to D. A run $r \in R$ *covers* c, if r fulfills c. Standard coverage tools determine the following sets:

$$\begin{aligned}
\textit{coveredBy(r)} &= \{c \in \mathcal{C} | r \text{ covers } c\} && \text{coverage items covered by } r \\
\textit{uncoveredBy(r)} &= \mathcal{C} \backslash \textit{coveredBy(r)} && \text{coverage items not covered by } r \\
\textit{coveredBySet}(R_s) &= \bigcup_{r \in R_s} \textit{coveredBy}(r) && \text{coverage items covered by runs in } R_s \\
\textit{uncoveredBySet}(R_s) &= \mathcal{C} \backslash \textit{coveredBySet}(R_s) && \text{coverage items not covered by } R_s \\
\textit{hit}(c) &= \{r \in R | r \text{ covers } c\} && \text{runs covering } c \\
\textit{miss}(c) &= R \backslash \textit{hit}(c) && \text{runs not covering } c
\end{aligned}$$

with $r \in R$, $R_s \subseteq R$, and $c \in \mathcal{C}$.

4. Locating Features

In this section we present our approach for feature localization in hardware designs. The main idea of feature localization using coverage metrics is to compare the coverage of runs for a system which use a certain feature with those not using this feature. Therefore several different runs of the system under test are required. An underlying assumption for our approach is, that for a developer it is easier to decide whether a run is related to a feature than deciding whether a coverage item is related to a feature. Which feature $f \in F$ is used by a run must be either specified by a developer or taken from the test bench documentation:

$$\begin{aligned} \mathit{use}(f) &= \{r \in R | r \text{ uses } f\} && \text{runs using } f \\ \mathit{notuse}(f) &= R \backslash \text{use}(f) && \text{runs not using } f \end{aligned}$$

Based on the user input and the coverage information the relation between features and coverage items is computed:

$$\begin{aligned} \mathit{pass}(c, f) &= \mathit{hit}(c) \cap \mathit{use}(f) && \text{runs covering } c \text{ and using } f \\ \mathit{fail}(c, f) &= \mathit{hit}(c) \cap \mathit{notuse}(f) && \text{runs covering } c \text{ and not using } f \end{aligned}$$

A coverage item c is likely related to a feature f for example, if c is covered whenever f is used but never covered when f is not used, or formally: $(\mathit{pass}(c, f) \equiv \mathit{use}(f)) \wedge (\mathit{fail}(c, f) \equiv \emptyset)$. Still the difference in the coverage may have other reasons. For some coverage item c it might be possible that $c \notin \mathit{coveredBySet}(\mathit{use}(f))$, even though c is related to the implementation of f. For example, if c is related to a special case of f. Having only small differences between the runs, which use a feature and which do not, as well as having runs which use as few other features as possible often improves the result [Fan02].
There exist several different heuristics for computing the likelihood of a coverage item to be related to a feature, which base on the coverage information. Section 4.1 presents four heuristics adapted from literature. Section 4.2 introduces an additional heuristic and discusses the different heuristics and their relation to each other. Finally, Section 4.3 presents the prototype implementation of our technique.

4.1. Coloring heuristics

For feature localization, there exist several heuristics to relate the source code parts to a feature [EKS03, WS95]. For evaluating which heuristics are best for the localization of features in hardware designs, four heuristics from literature have been adapted for our technique. To present the results to the user we use color coding. This way of presentation is inspired by the *Tarantula* tool [JHS01].
In [EKS03] two different categorizations for feature localization are described. Both categorizations are defined over a set of computational units. Based on how fine-grained the partition should be, a computational unit can be for example a source code statement, a basic block, or a function. For our approach we define the categorizations over the set of coverage items $\mathcal{C}$. The first categorization (Cat_1) partitions the coverage items from $\mathcal{C}$, which were covered by at least one run using the wanted feature $f \in F$, into four different categories defined as:

1. Coverage items covered if and only if f is used:
$$\mathit{uniquelyInvolved}(f) = \{c \in \mathcal{C} | \ (\mathit{pass}(c, f) \equiv \mathit{use}(f)) \wedge (\mathit{fail}(c, f) \equiv \emptyset)\}$$

2. Coverage items always covered when f is used and sometimes when f is not used:
$$\mathit{indispensablyInvolved}(f) = \{c \in \mathcal{C} |\ (\mathit{pass}(c, f) \equiv \mathit{use}(f)) \wedge (0 < |\mathit{fail}(c, f)| < |\mathit{notuse}(f)|)\}$$
3. Coverage items sometimes covered when f is used:
$$\mathit{potentiallyInvolved}(f) = \{c \in \mathcal{C} |\ 0 < |\mathit{pass}(c, f)| < |\mathit{use}(f)|\}$$
4. Coverage items always covered:
$$\mathit{commonlyInvolved}() = \{c \in \mathcal{C} |\ \forall r \in R, c \in \mathit{coveredBy}(r)\}$$

As this categorization only considers a coverage item *c* if *pass(c,f)* $\neq \emptyset$, there is a fifth implicit category:

5. Coverage items never covered when f is used:
$$\mathit{unconsidered}(f) = \{c \in \mathcal{C} |\ \mathit{pass}(c, f) \equiv \emptyset\}$$

The second categorization (Cat_2) partitions the coverage items with respect to a certain feature into five groups defined as:

1. Coverage items covered if and only if f is used:
$$\mathit{specific}(f) = \{c \in \mathcal{C} |\ (\mathit{pass}(c, f) \equiv use(f)) \wedge (\mathit{fail}(c, f) \equiv \emptyset)\}$$
2. Coverage items always covered when f is used and at least once when f is not used:
$$\mathit{relevant}(f) = \{c \in \mathcal{C} |\ (\mathit{pass}(c, f) \equiv use(f)) \wedge (\mathit{fail}(c, f) \neq \emptyset)\}$$
3. Coverage items sometimes covered when f is used and never when f is not used:
$$\mathit{conditionalSpecific}(f) = \{c \in \mathcal{C} |\ 0 < |\mathit{pass}(c, f)| < |\mathit{use}(f)| \wedge (\mathit{fail}(c, f) \equiv \emptyset)\}$$
4. Coverage items sometimes covered when f is used and at least once when f is not used:
$$\mathit{shared}(f) = \{c \in \mathcal{C} |\ 0 < |\mathit{pass}(c, f)| < |\mathit{use}(f)| \wedge (0 < |fail(c, f)|)\}$$
5. Coverage items never covered when f is used:
$$\mathit{irrelevant}(f) = \{c \in \mathcal{C} |\ \mathit{pass}(c, f) \equiv \emptyset\}$$

In addition to the presented categorizations, two coloring schemes from bug localization in software are adapted for our approach. The first scheme extends the two-dimensional *Tarantula* scheme [JHS01] to differentiate multiple features. The first dimension is the likelihood like_T of c being related to a feature f:

$$\mathit{like}_T(c, f) = \frac{\mathit{passed}(c, f)}{\mathit{passed}(c, f) + \mathit{failed}(c, f)}$$

with $\mathit{passed}(c, f) = \frac{|\mathit{pass}(c,f)|}{|\mathit{use}(f)|}$ and $\mathit{failed}(c, f) = \frac{|\mathit{fail}(c,f)|}{|\mathit{notuse}(f)|}$. To prevent undefined values let $\mathit{like}_T(c, f) = 0$ if $\mathit{hit}(c) \equiv \emptyset$. Our formula is a generalization of the original formula. The value of the original formula can be computed by fixing the feature f to "not passes the test case". For our approach, the hue of a coverage item is defined by its likelihood. The hue reaches from green ($\mathit{like}_T = 1$) over yellow ($\mathit{like}_T = 0.5$) to red ($\mathit{like}_T = 0$). The second dimension of the *Tarantula* scheme estimates the confidence *con* towards the likelihood value of c. The confidence is defined as:

$$\mathit{con}(c, f) = \mathit{max}(\mathit{passed}(c, f), \mathit{failed}(c, f))$$

The confidence is visualized as the brightness in which c is colored. The brightness of c is linear to its confidence. A high confidence ($con = 1$) is colored brightest and a low confidence ($con = 0$) is colored darkest.
The other coloring scheme from bug localization in software uses the *Ochiai coefficient* for computing the likelihood. Compared to the *Tarantula* scheme, the *Ochiai* coloring scheme yields better results in case of bug localization in software [AZvG07]. Again we generalize the formula by adding a parameter for the wanted feature f, such that we compute the likelihood $like_O$ of a coverage item c to be related to f by:

$$like_O(c, f) = \frac{|pass(c, f)|}{\sqrt{|use(f)| * |hit(c)|}}$$

To prevent undefined values let $like_O(c, f) = 0$ if $hit(c) \equiv \emptyset$. The computation of the confidence is identical to the *Tarantula* scheme.

4.2. Discussion

Early experiments with Cat_2 have shown that for hardware designs a large portion of the coverage items is categorized as *relevant* even if it has nothing to do with the feature. This is due to the fact that there is much code which is always executed, like *always-blocks* and *assign-statements*. For overcoming these problems we propose an extension of Cat_2. This extension Cat_{ext} introduces the category *common* and redefines *relevant* as:

2a. Coverage items always covered:
$$common(f) = \{c \in \mathcal{C} |\ \forall r \in R, c \in coveredBy(r)\}$$

2b. Coverage items always covered when f is used and sometimes covered when f is not used:
$$relevant_{ext}(f) = \{c \in \mathcal{C} |\ (pass(c, f) \equiv\ use(f)) \wedge (0 < |fail(c, f)| < |notuse(f)|)\}$$

The three categorizations and the *Tarantula* scheme are related to each other. Categorization Cat_{ext} subsumes Cat_1 and Cat_2. This means, it is possible to use Cat_{ext} to compute Cat_1 and Cat_2. Computing Cat_1 from Cat_2 or vice versa, however, is not possible. The *Tarantula* scheme subsumes all three categorizations. By partitioning the *Tarantula* scheme in different classes the categorizations can be computed. Table 1 shows the relation between the four schemes and describes which color is used for which category. The *Ochiai* scheme cannot be related to the other coloring schemes, because this scheme also considers the total number of runs covering c.
As an advantage the *Tarantula* and the *Ochiai* scheme provide a continuous range preventing runs with very high or very low coverage to have a disproportionately strong effect on the result. Both schemes also identify coverage items that are not covered if and only if a certain feature is used. The other three do not distinguish between not covered items while using a feature f, and items never covered.

4.3. Prototype

Our prototype applies the coloring schemes and presents them to a user. The current implementation considers line coverage and toggle coverage for the visualization. Thus, on the one hand we have a

Table 1: The relation between the different categorizations and the Tarantula coloring scheme and the colors used for the different categorizations

Cat_1		Cat_2		Cat_{ext}		Tarantula
Category	Color	Category	Color	Category	Color	equivalent class
uniquelyInvolved	bright green	*specific*	bright green	*specific*	bright green	$like_T = 1 \wedge$ $con = 1$
commonly-Involved	bright yellow	*relevant*	yellow green	*common*	bright yellow	$like_T = 0.5 \wedge$ $con = 1$
indispensably-Involved	dark green			*relevant*$_{ext}$	yellow green	$0.5 < like_T < 1 \wedge$ $con = 1$
potentially-Involved	yellow green	*conditional-Specific*	dark green	*conditional-Specific*	dark green	$like_T = 1 \wedge$ $con < 1$
		shared	dark yellow	*shared*	dark yellow	$(con = 1 \wedge$ $0 < like_T < 0.5) \vee$ $(0 < con < 1)$
unconsidered	dark grey	*irrelevant*	dark grey	*irrelevant*	dark grey	$like_T = 0$

basic coverage metric often used for feature localization in software. On the other hand we have a coverage metric which is unique to hardware design. Because it is not trivial to relate toggle coverage to statements our current prototype computes and colors the different coverage metrics separately. Combining several coverage metrics like in the work of Santelices et. al. [SJYH09], will be an aspect of future work. Currently, the supported HDL is Verilog. *ModelSim* [Mod] is used to collect the coverage information in form of *Unified Coverage Data Base* (UCDB)-files.
Figure 1 shows screenshots of our prototype. At the top left the user can choose which feature of the design shall be inspected. At the top right the coloring schemes can be chosen. Note that the user may use different coloring schemes for statement coverage and toggle coverage. Below the user can choose the sourcecode file for inspection. For each source code file there is a tab. In the source code view below each line is colored with respect to its line coverage. Toggle coverage is represented by over- and underlining variables. Overlining represents the toggle coverage from 0 to 1 of a variable and underlining the toggle coverage from 1 to 0. Also each line is annotated with its corresponding line number.

5. Case Studies

In order to evaluate our approach, we considered designs that have to fulfill the following requirements: they provide several different features, they are written in Verilog, the designs and the corresponding test benches run in *ModelSim*, and they have a well commented test bench either distinguishing the different features or allowing to easily use the test bench as template for use cases. Two designs from the *OpenCores* website [Ope], fulfilling these requirements, are chosen.
When we conducted our case study we used our prototype to gather information about the implementations of the different features of the designs. Initially, the only information we had about these designs was their brief description at *OpenCores*, the file name of the different Verilog files and knowledge about the design of the test bench. While using our prototype we documented our findings and which coverage metric and coloring scheme helped us to obtain them. After we were finished with inspecting the design, we checked our findings against the documentation of the design.
Table 2 gives a brief overview of the designs used for the case studies. The column *Design* contains

Table 2: Overview of the designs used in the case studies

Design	LOC	Files	use cases	features	computation time
double_fpu_verilog	2555	7	144	8	22.4 sec
SD/MMC Controller	3840	17	5	5	2.8 sec

the title under which the designs are listed at the *OpenCores* website. Lines of code (*LOC*) is the number of all non-comment and non-empty lines of the source files. The column *computation time* shows the time the prototype required to compute all coloring schemes for the given design from the coverage data. The results show that computing the coloring schemes is fast once the coverage data is gathered. In many cases this coverage information will already be computed during the validation of the design. Also in cases where the coverage information must be computed, the different use cases can be simulated in parallel, reducing the overall time of gathering the coverage information.

5.1. Case Study: double_fpu_verilog

This case study considers a double precision FPU which requires 20 (addition) to 71 (division) clock cycles per operation. The supported features are four arithmetic operations:

- `addition`
- `subtraction`
- `multiplication`
- `division`

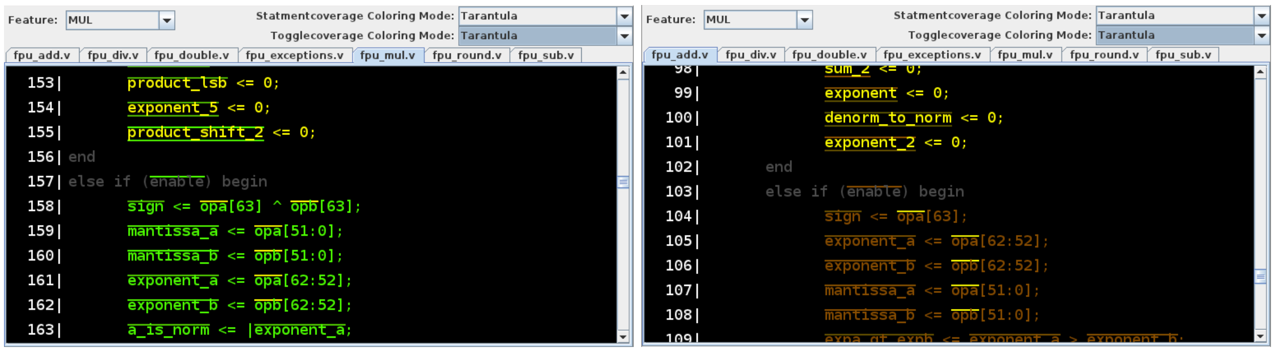
and four rounding modes:

- `round to nearest even`
- `round to zero`
- `round to +INF`
- `round to -INF`

For each combination of operation and rounding mode, there exist nine use cases. The documentation consists of a pdf-file with twelve pages and very few source-code comments.

There is a huge difference between the difficulty to localize arithmetic operations and to localize rounding modes. For the arithmetic operations, the statement-coverage-based coloring schemes provide several locations related to the feature. But still 56% of the statements are executed for all use cases. For these statements, statement coverage cannot help to decide whether they are relevant for the feature or not. The information provided to the user based on statement coverage is identical for all coloring schemes such that no qualitative difference can be found between them.

Toggle coverage allows to partition the statements always executed in statements that use toggling signals and statements using signals that do not toggle. Since statements that use constant signals are unlikely to be part of the computation, they can be filtered out. When considering toggle coverage all coloring schemes support the user while locating features. However, the *Tarantula* coloring scheme provides the strongest contrast and therefore shows the difference in toggle coverage very clearly. The *Ochiai* scheme also provides the information clearly, but with less contrast, making it harder to recognize. These two coloring schemes show that a coverage item is not covered if and only if a feature is executed. When relating this information to toggle coverage, this translates to a given register or wire staying constant if and only if a given feature is used. This information helps to understand a feature, as already assumed in Section 4.2. The three other schemes do not provide this information.



Statmentcoverage Coloring Mode: Tarantula
Togglecoverage Coloring Mode: Tarantula

```
        product_lsb <= 0;
        exponent_5 <= 0;
        product_shift_2 <= 0;
end
else if (enable) begin
        sign <= opa[63] ^ opb[63];
        mantissa_a <= opa[51:0];
        mantissa_b <= opb[51:0];
        exponent_a <= opa[62:52];
        exponent_b <= opb[62:52];
        a_is_norm <= |exponent_a;
```

```
                sum_2 <= 0;
                exponent <= 0;
                denorm_to_norm <= 0;
                exponent_2 <= 0;
        end
        else if (enable) begin
                sign <= opa[63];
                exponent_a <= opa[62:52];
                exponent_b <= opb[62:52];
                mantissa_a <= opa[51:0];
                mantissa_b <= opb[51:0];
                expa_gt_expb <= exponent_a > exponent_b;
```

Figure 1: Screenshots of our prototype inspecting a part of the design related to a feature, as claimed by the documentation (left), and a part of the design which are not related to the feature, as claimed by the documentation (right)

Figure 1 gives two examples presenting the FPU design to the user. The design is partitioned in case of the `multiplication` operation. For this example statement coverage provides a clear distinction for some parts of the design. Toggle coverage provides additional information to further distinguish statements always executed (yellow statements).
More difficult is the localization of the rounding features. Based on statement coverage there is no difference between the rounding modes, forcing the user to completely rely on toggle coverage. Even for toggle coverage there is only a small difference. In case of `round to nearest even` only the *Tarantula* or the *Ochiai* scheme show a difference. For identifying the feature it is necessary to compare the coloring with other rounding modes. Altogether in case of the rounding modes, the feature localization results in two to four statements corresponding to each rounding mode.
Comparing our findings with the documentation shows the advantages of our approach. First the documentation only describes in which module a feature is implemented. All the positions found with our approach are placed in the corresponding module. In conclusion, our approach retrieves results that are at least as good as those contained in the documentation. Moreover, there are special cases for `addition` and `subtraction` based on the signs of the operands. An `addition` may be executed by the subtraction unit and vice versa. The documentation does not include this information in the description of the two operations, but in the description of the design hierarchy. By this, someone only reading the operation descriptions would miss this peculiarity. Additionally, our approach highlights the signal that defines which variant is used. This information does not even exist in the documentation. After inspecting the corresponding code for the rounding modes we are confident that the lines marked by the prototype are in fact the main parts implementing the rounding features.
In conclusion, the *Tarantula* coloring scheme has provided the best results and statement coverage gives a first overview, where toggle coverage allows to further differentiate those statements which always are executed. The arithmetic operations were practically found at first glance, and except for the `round to nearest even` all features were found faster than by looking at the documentation. Finally, our prototype yields more information about the design than the documentation does.

5.2. Case Study: SD/MMC Controller

The design used for this case study is a controller chip for SD/MMC cards for up to 2GB. The controller is accessed through a Wishbone-slave-interface. The test bench of the controller includes

Table 3: The definitions for the likelihood of a file to be related to a feature

Likelihood	Description
+++	More than 25% of statements colored bright green.
++	Statements colored bright green but less than 25%
+	No green statements but green toggles
0	Yellow toggles and statements
-	Yellow statements, most with yellow toggles, few with red toggles
–	Yellow and red statements
—	Red and yellow statements, including bright red statements

a Wishbone simulator and an SD-card simulator used for testing. The five features as defined by the test bench are:

- `Register access`
- `SPI bus access`
- `SD init`
- `SD write`
- `SD read`

The test bench consists of only one function but clearly defines which feature is used. we used this distinction to measure the different coverages for the corresponding executions. The documentation of the controller consists of two pdf-files, one consisting of 23 pages and the other one consisting of 17 pages. Additionally, there are several source code comments.

In contrast to the first case study, all features are equally easy to find in the SD/MMC Controller. The realated source code is less easy to spot than the arithmetic operations in the first case study, but easier than the rounding modes. When comparing the different coloring schemes we observed that the run for the `Register access` feature covers very few coverage items, resulting in the effect that the categorization based schemes mark the coverage items which are covered by all the other runs as *relevant* or *indispensablyInvolved*, respectively. This practically renders the categorization schemes useless and is very similar to the effect which caused us to introduce Cat_{ext}. However, the *Tarantula* scheme and the *Ochiai* scheme are not compromised. The computed likelihood is only minimally affected by such use cases, because of the continuous range of the schemes. The *Tarantula* scheme and the *Ochiai* scheme obtain good results, with no visible differences.

Because of the experience from the first case study that the feature implementation was not well documented, we assign each source code file a likelihood to be related to the feature based on the results of our technique. Those likelihoods are described in Table 3.

After we finished our inspection we checked the documentation to find out where which feature is implemented. The pdf-files of the documentation did not help because they only explain how to use the design. However, most of the source code files have a description explaining their purpose. In many cases this description can directly be related to a feature. However, there are some files without any description, e.g. *sm_RxFifoBl.v*, or files where the description could not be related to any feature, e.g. *sm_fifoRTL.v*. Additionally, as the design is accessed through a Wishbone-interface the files related to the Wishbone-interface are considered as commonly used by all features.

Table 4 compares our findings with the claims of the documentation. The column *Most Likely* shows the files which we assigned the highest likelihood and the column *Second Likely* shows those files with the second highest likelihood. In case of the feature `SPI bus access` there are no files annotated with +++, +, or 0 such that the highest likelihood contains all files with ++ and the second highest likelihood those with -. If the documentation clearly relates a file to a feature the file is listed

Table 4: Likelihood for the source code files compared to the documentation. The percentage given is the percentage of statements colored as required by the definition of their likelihood, for +++ and ++ this are bright green statements, for - this are yellow statements with yellow toggles

Feature	*Tarantula*/*Orchiai* scheme						documentation	
	Most Likely			Second Likely			Belongs	Possible
Register access	+++	sm_TxFifoBl[*] ctrlStsRegBl[†]	34% 36%	++	sm_fifoRTL[*] sm_RxFifoBl[*]	17% 20%		
SD init	+++	initSD	39%	++	spiCtrl	14%	initSD spiCtrl	sendCMD
SD read	+++	readWriteSDBlock	31%	++	sm_RxFifoBl[*] spiCtrl spiMasterWishBoneBl[†]	20% 12% 24%	readWriteSDBlock spiCtrl	sendCMD
SD write	+++	readWriteSDBlock sm_TxFifoBl[*] sm_fifoRTL[*]	41% 68% 58%	++	spiMasterWishBoneBl[†] spiCtrl	12% 14%	readWriteSDBlock spiCtrl	sendCMD
SPI bus access	++	ctrlStsRegBl[†] spiCtrl spiTxRxData	4% 18% 9%	-	readWriteSPIWireData	64%	readWriteSPIWireData	spiCtrl spiTxRxData

in column *Belongs* and those files where the documentation is unclear are listed in column *Possible*. Columns *Most Likely* and *Second Likely* include all files the documentation relates to the feature. For all additionally included files the documentation either relates them to all features, or does not contain information about them, or is unclear regarding the feature. Again our approach yields information that is as good as the documentation. Within each file our approach partitions the different parts of the HDL source very clearly, as the percentages in Table 4 show. Because of the poor documentation, we cannot verify this partitioning.
This case study shows clearly that the three categorization based coloring heuristics are inaccurate when being faced with a single use case yielding very low coverage. The *Tarantula* and the *Ochiai* scheme provide equally good results as there are no visible differences between both schemes. Again, our approach gives at least as good information as the documentation. Moreover, the approach often provides additional information for feature localization. Therefore techniques for feature localization, like the one presented in this paper, are needed for design understanding.

6. Conclusion

We described an approach for feature localization in hardware designs. Our approach uses coverage information gathered by simulation to relate different coverage items to different features. The current implementation only supports statement coverage and toggle coverage. Five different coloring schemes to present the results have been implemented. Three categorize the coverage items into different groups. The other two schemes compute the likelihood of a coverage item to be related to a feature and the confidence in this likelihood. These values are then presented as the hue and the brightness of the coverage items.
The case studies emphasize the strength of our approach. They also showed that the *Tarantula* coloring scheme performs best and that coverage metrics typically used for feature localization in software systems are not sufficient for feature localization in hardware designs. Therefore hardware specific

[*]File that is not documented or the documentation does not relate it to any feature
[†]File that the documentation claims to belong to the Wishbone-interface and therefore is commonly used for all features

coverage metrics must be used as well. Additionally the two case studies showed that the main advantages of the *Tarantula* and *Ochiai* scheme are their continuous range and their notion of not covering a coverage item if a certain feature is used.

References

[AZvG06] Abreu, R., P. Zoeteweij, and A. J. C. van Gemund: *An evaluation of similarity coefficients for software fault localization*. In *Pacific Rim International Symposium on Dependable Computing*, pages 39 –46, 2006.

[AZvG07] Abreu, R., P. Zoeteweij, and A. J. C. van Gemund: *On the accuracy of spectrum-based fault localization*. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89 –98, 2007.

[EKS03] Eisenbarth, T., R. Koschke, and D. Simon: *Locating features in source code*. IEEE Transactions on Software Engineering, 29:210–224, 2003.

[Fan02] Fantozzi, A.: *Locating features in vim: A software reconnaissance case study*. Technical report, 2002.

[IEE06] IEEE 1364 Working Group: *IEEE Standard for Verilog Hardware Description Language*. IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001), 2006.

[ITR09] ITRS Working Group: *International technology roadmap for semiconductors 2009 update system drivers*, 2009. `http://www.itrs.net`.

[JHS01] Jones, J. A., M. J. Harrold, and J. T. Stasko: *Visualization for fault localization*. In *Proceedings of the Workshop on Software Visualization*, pages 71 –75, 2001.

[Mod] *ModelSim Website*. `http://model.com/`.

[Ope] *OpenCores Website*. `http://opencores.org/`.

[SJYH09] Santelices, R., J. A. Jones, Y. Yu, and M. J. Harrold: *Lightweight fault-localization using multiple coverage types*. In *International Conference on Software Engineering*, pages 56–66, 2009.

[TK01] Tasiran, S. and K. Keutzer: *Coverage metrics for functional validation of hardware designs*. Design Test of Computers, IEEE, 18(4):36 –45, 2001.

[WC96] Wilde, N. and C. Casey: *Early field experience with the software reconnaissance technique for program comprehension*. In *Working Conference on Reverse Engineering*, pages 270 –276, 1996.

[WS95] Wilde, N. and M. C. Scully: *Software reconnaissance: Mapping program features to code*. Journal of Software Maintenance: Research and Practice, 7(1):49–62, 1995.