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

Jan Malburg[*]
malburg@informatik.uni-bremen.de

Alexander Finder[*]
final@informatik.uni-bremen.de

Görschwin Fey[*][†]
Goerschwin.Fey@dlr.de

[*]University of Bremen
28359 Bremen, Germany

[†]German Aerospace Center
28359 Bremen, Germany

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 fast and more precise than the existing documentation for the inspected designs.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids

General Terms

Design, Documentation, Experimentation

Keywords

Feature Localization, Design Understanding, Simulation

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 [9] 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 a result becomes incorrect, or performance, limiting the time until a design has to return the expected output. In the following only functional features are considered and for simplicity called features.
For design improvement, design extension, and bug fixing a developer has to understand the design. These tasks become even more important, since the amount of re-used design blocks is continuously increasing in future [9]. 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 new approach for locating parts of the HDL code which are relevant for a functional feature. The proposed technique uses a dynamic approach relating coverage information gathered by simulation to features executed by use cases. This technique is basically usable with all HDLs and representations given the design can be simulated and coverage can be measured, e.g. Register-Transfer-Level or Transaction-Level descriptions. We currently support Verilog in our prototype. 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,
- an adaptive ranking of source files according to their likelihood to be related to a feature,
- a comparison of orthogonal features to improve design understanding,

Experiments showed that, our approach even provides good results when applied to designs with poorly separated features.

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

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

2. RELATED WORK

Techniques for design understanding of HDL descriptions concentrated on inferring specifications from traces [7, 4, 11] or merging partial specifications to more abstract ones [13]. Another technique is program slicing [3], which differs from feature localization as it answers the questions which parts of the code can affect or be affected by a signal. Instead feature localization answers the question which parts of the code are responsible for creating a defined output under certain input. So far, no technique has been published about feature localization in HDL descriptions.

Feature localization for software designs is an active research area. Often the statement coverage of runs using a wanted feature is compared to the coverage of runs not using this feature [16]. 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 [15]. 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 [5].

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* [10]. *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 which execute a statement to non-failing runs executing the statement. Abreu et. al. [1] 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. [12] presented an approach to relate branch coverage 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 [8]. Finally, hardware designs are inherently parallel.

3. PRELIMINARIES

In this section we provide some 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 $i_j, j = 1, ..., m$ 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 run r is the simulation of D applying a use case. 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 over elements in R. A *coverage item* $c \in C$ is a single condition. The form of these conditions is defined by C. A feature f is a distinguishing characteristic of D, defining the expected output of D under specific input. The set $F = \{f_1, f_2, f_3, .., f_k\}$ is the set of all features supported by D. The user defines, if a run r *uses* a feature f. A feature f is implemented by a set of coverage items C_f. The goal of feature localization is to determine C_f.

Two typical coverage metrics used in hardware design are statement coverage and toggle coverage [14]. Toggle coverage is a coverage metric especially for hardware design. Statement coverage is also used in software design [10]. 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:

1. Coverage items covered by r:
   $$\textit{coveredBy(r)} = \{c \in \mathcal{C} | r \textit{ covers } c\}$$
2. Coverage items not covered by r:
   $$\textit{uncoveredBy(r)} = \mathcal{C} \setminus \textit{coveredBy(r)}$$
3. Coverage items covered by R_s:
   $$\textit{coveredBySet}(R_s) = \bigcup_{r \in R_s} \textit{coveredBy}(r)$$
4. Coverage items not covered by R_s:
   $$\textit{uncoveredBySet}(R_s) = \mathcal{C} \setminus \textit{coveredBySet}(R_s)$$
5. Runs covering c:
   $$\textit{hit}(c) = \{r \in R | r \textit{ covers } c\}$$
6. Runs not covering c:
   $$\textit{miss}(c) = R \setminus \textit{hit}(c)$$

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

4. LOCATING FEATURES

In this section we will 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 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 if a run is related to a feature than deciding if a coverage item is related to a feature. Which feature $f \in F$ is used by a run must either be specified by a developer or taken from the test bench documentation: Based on the user input and the coverage informa-

7. Runs using f:
   $$\textit{use}(f) = \{r \in R | r \textit{ uses } f\}$$
8. Runs not using f :
   $$\textit{notuse}(f) = R \setminus \textit{use}(f)$$

tion the relation between features and coverage items is computed: Intuitively, a coverage item c is likely related to a feature f, if c is

9. Runs covering c and using f:
   $$\textit{pass}(c, f) = \textit{hit}(c) \cap \textit{use}(f)$$
10. Runs covering c and not using f:
   $$\textit{fail}(c, f) = \textit{hit}(c) \cap \textit{notuse}(f)$$

covered whenever f is used but never covered when f is not used, or formally: $(\textit{pass}(c, f) \equiv \textit{use}(f)) \wedge (\textit{fail}(c, f) \equiv \emptyset)$. Still the difference in the coverage may have other reasons. For some coverage item c it might be possible that $c \notin \textit{setCoveredBy}(\textit{use}(f))$, even though c is related to the implementation of f. For instance, 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 [6]. Next we will present three coverage based heuristics for computing the likelihood of a coverage item to be related to a feature.

4.1 Coloring heuristics

For feature localization, there exist several heuristics to relate the source code parts to a feature [5, 16]. For evaluating which heuristics are best for the localization of features in hardware designs, three 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 [10].

In [5] a categorization for feature localization is described. This categorization is 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 categorization over the set of coverage items $\mathcal{C}$. This categorization (Cat) partitions the coverage items with respect to a certain feature into five groups defined as: In addition to the presented categorization, two coloring

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

schemes from bug localization in software are adapted for our approach. The first scheme extends the two-dimensional *Tarantula* scheme [10] to differentiate multiple features. One dimension is the likelihood $\textit{like}_T$ of a coverage item c to be related to feature f:

$$\textit{like}_T(c,f) = \begin{cases} \frac{\textit{passed}(c,f)}{\textit{passed}(c,f)+\textit{failed}(c,f)} & \text{if } \textit{hit}(c) \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

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

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

The confidence is visualized as the brightness in which c is colored. The brightness of c is linear to its confidence. The highest confidence ($\textit{con} = 1$) is colored brightest and the lowest confidence ($\textit{con} = 0$) is colored darkest.

The other coloring scheme adapted 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 [2]. Again we generalize the formula by adding a parameter for the wanted feature f, such that we compute the likelihood $\textit{like}_O$ of a coverage item c to be related to f by:

$$\textit{like}_O(c,f) = \begin{cases} \frac{|\textit{pass}(c,f)|}{\sqrt{|\textit{use}(f)| * |\textit{hit}(c)|}} & \text{if } \textit{hit}(c) \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

The computation of the confidence is identical to the *Tarantula* scheme.

4.2 Comparison

Early experiments with Cat 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. This extension Cat_{ext} introduces the category *common* and redefines *relevant* as: The categorizations

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

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

and the *Tarantula* scheme are related to each other. Categorization Cat_{ext} subsumes Cat and the *Tarantula* scheme subsumes the categorizations. By partitioning the *Tarantula* scheme in different classes the categorizations can be computed. Table 1 shows the relation between the three 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 two do not distinguish between not covered items while using a feature f, and items never covered.

4.3 Feature comparison

Often there are sets of features for which at a point in time at most one feature in the set can be used. Such features are called *orthogonal*. The user can define features as *orthogonal* to each other. An extension unique to our approach is the comparison of two *orthogonal* features. This allows the user to see which are the parts where the features differ from each other and therefore gives additional insight to the implementation of the features and the design as a whole. This comparison is defined over the likelihood and the confidence of the features and therefore only usable for the *Tarantula* and *Ochiai* coloring schemes. The comparison value *comp* computes how likely a coverage item c is covered by one feature f_b but not by a feature f_c orthogonal to f_b. The comparison value is defined as:

$$\textit{comp}(c, f_b, f_c) = \frac{(1 + (\textit{like}(c, f_b) - \textit{like}(c, f_c))}{2}$$

with $\textit{like} \in \{\textit{like}_T, \textit{like}_O\}$ defining which coloring scheme is used for the comparison. The feature $f_b \in F$ is the feature which we want to inspect and $f_c \in F$ is the orthogonal feature which we want to compare with f_b. The mapping of the comparison value to hue is equivalent to mapping the likelihood to the hue. For the brightness, the maximum of the confidences is used:

$$\textit{brightness}(c, f_b, f_c) = \textit{max}(\textit{con}(c, f_b), \textit{con}(c, f_c))$$

4.4 File ranking

Another extension particular to our approach provides additional guidance for feature localization by ranking the different files based on their likelihood to be related to a feature. As the mapping of signals to files is a non-trivial task, our current implementation only considers statement coverage for the ranking. The ranking works

Table 1: Relation between categorizations and Tarantula coloring scheme; colors encoding categories

Cat		Cat_{ext}		Tarantula
Category	Color	Category	Color	equivalent class
specific	bright green	*specific*	bright green	$like_T = 1 \wedge con = 1$
relevant	yellow green	*common*	bright yellow	$like_T = 0.5 \wedge con = 1$
		relevant$_{ext}$	yellow green	$0.5 < like_T < 1 \wedge con = 1$
conditional	dark green	*conditional*	dark green	$like_T = 1 \wedge con < 1$
shared	dark yellow	*shared*	dark yellow	$(con = 1 \wedge 0 < like_T < 0.5) \vee (0 < con < 1 \wedge like_T < 1)$
irrelevant	dark grey	*irrelevant*	dark grey	$like_T = 0$

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

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

as follows: Initially the user has to choose a threshold for the computation. In case of the categorization this is a category and in case of the *Tarantula* and *Ochiai* scheme this is a minimum value for the likelihood. Then starting with the highest value (specific or likelihood of 1, respectively) as the upper bound and the lower bound, all files having statements within these bounds are considered and then ordered based on the percentage of statements within these bounds. Those files are added to the ranking in this order. The lower bound is reduced until more files are found or the given threshold is reached. In case a new file is found, it is added to the ranking. If several files are found at the same time, they are added to the ranking ordered by the percentage of statements within the bounds.

5. CASE STUDIES

For testing our approach we have implemented a prototype. This prototype uses *ModelSim*, to compute the coverage of the different use cases. The current version of our prototype supports only Verilog, but this is only a technical limitation of our prototype. Adding the support for additional HDLs requires only adding an additional parser, which can translate the hierarchical signal name to the local signal names in each source code file. In the current version our implementation supports statement and toggle coverage, where statement coverage is represented by coloring the corresponding lines. Toggle coverage is represented by overlining, in case of a toggle from 0 to 1, or underlining, in case of a toggle from 1 to 0, the corresponding signals.

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 website *OpenCores.org*, fulfilling these requirements, have been chosen. We conducted our case study as follows:

1. we looked for a design, unknown to us, which provides several features and including a test bench testing those features,
2. we analyzed the design using our prototype,
3. we wrote down all our findings,
4. finally, we checked our findings against the documentation.

Note, as we only used designs which were originally unknown to us, the only information we had about the designs were their descriptions at *OpenCores.org* and the structure of their test benches. Table 2 gives a brief overview of the designs used for the case studies. The column *Design* contains the title under which the designs are listed at *OpenCores.org*. Lines of code (*LOC*) is the number of all non-comment and non-empty lines of the design. In column *time* the time required to compute and present the heuristics is shown. Compared to the time for simulation and coverage gathering, which takes 30 minutes for double_fpu_verilog and 18 seconds for SD/MMC Controller, the computation of the heuristic is rather fast, making the simulation the main limitation of our technique. In many cases this coverage information will already be computed during the validation of the design. In our studies, gathering the coverage information has not increased the time needed for the simulation, i.e. the computational overhead of our approach is negligible.

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:

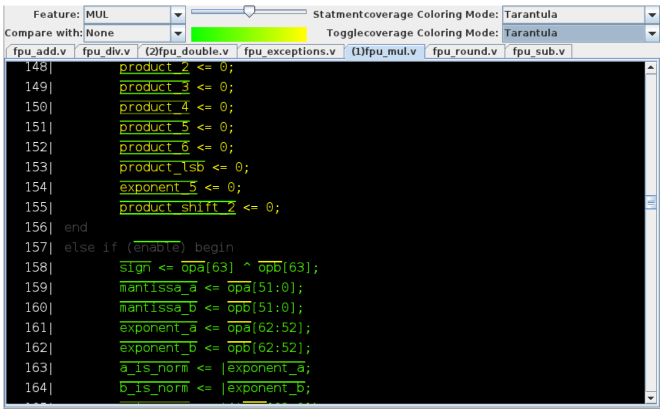
- `round to nearest even`
- `round to zero`
- `round to +INF`
- `round to -INF`

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 part of the feature or not. The information provided to the user based on statement coverage is very similiar for all coloring schemes such that no qualitative difference can be found between them.

When in addition considering toggle coverage, it is easy to partition the statements always executed in statements that use toggling signals and statements using not toggling signals. Since statements that operate on constant values are unlikely to be part of the computation, they can be filtered out. When considering toggle coverage all coloring schemes can support the user by locating features. 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 whether 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. As the two categorization schemes do not provide this information, it is not possible to recognize which signals are changing and which are not using them.

Figure 1 gives two examples how the FPU design is presented to

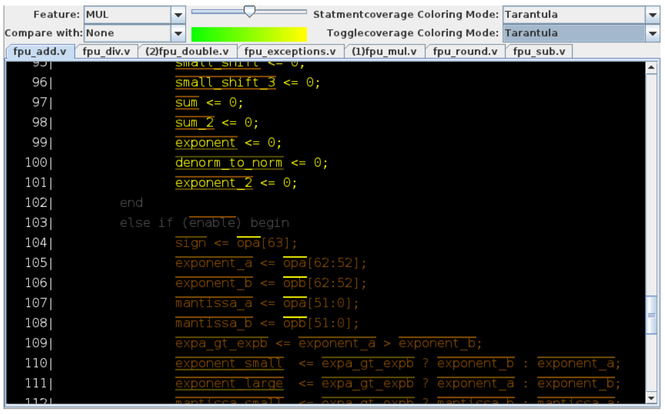

Feature: MUL

Compare with: None

Statmentcoverage Coloring Mode: Tarantula

Togglecoverage Coloring Mode: Tarantula

```
        product_2 <= 0;
        product_3 <= 0;
        product_4 <= 0;
        product_5 <= 0;
        product_6 <= 0;
        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;
        b_is_norm <= |exponent_b;
```


Feature: MUL

Compare with: None

Statmentcoverage Coloring Mode: Tarantula

Togglecoverage Coloring Mode: Tarantula

```
                small_shift_3 <= 0;
                sum <= 0;
                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;
                exponent_small  <= expa_gt_expb ? exponent_b : exponent_a;
                exponent_large  <= expa_gt_expb ? exponent_a : exponent_b;
```

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

Table 3: The file ranking for the arithmetic operations of the double_fpu_verilog design compared to the documentation.

Feature	*Tarantula* scheme	documentation
Addition	fpu_sub fpu_add	fpu_add fpu_sub
Substraction	fpu_double fpu_sub fpu_add	fpu_sub fpu_add
Multiplication	fpu_mul	fpu_mul
Division	fpu_div	fpu_div

the user, and how clearly the design is partitioned in case of an arithmetic operation (`multiplication`). The example shows that statement coverage provides a clear distinction for some parts of the design, but also that the 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 very little difference. In case of `round to nearest even` only the *Tarantula* or the *Ochiai* scheme show a difference, still for identifying the feature it is necessary to use the feature comparison functionality of our approach. Altogether in case of the rounding modes the feature localization results in 2-4 statements corresponding to each rounding mode. The comparison of our findings with the documentation shows the benefits of our approach. First the documentation only describes in which module a feature is implemented, and all the positions found with our approach are placed in the corresponding module. Therefore, we are able to get at least as good results as someone reading the documentation of the design. Also there are special cases for `addition` and `subtraction` based on the signs of the operands. An `addition` could 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 determines 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 in fact are the main parts implementing the rounding features. Again this is information not included in the documentation. The result of the comparison of the file ranking and the documentation is shown in Table 3. Only the file ranking for the *Tarantula* scheme is shown because this scheme yields the best results. As the file ranking currently only considers statement coverage only the arithmetic operations are shown. For the rounding mode all files have been included for each rounding mode. Similarly to the approach in [12] only the ranked files are shown until the point where all files are included which the documentation claims to belong to the feature. Except of for `subtraction` these are exactly those file which the documentation relates to the feature. In case of `subtraction` also the top-module is included as it contains some statements executed if and only if subtraction is used.
In conclusion, the *Tarantula* coloring scheme has provided the best results and statement coverage gives a first overview. Toggle coverage allows to differentiate those statements which are always executed. The arithmetic operations were practically found at the first glance, and except for the `round to nearest even` all features were found faster than by looking at the documentation. In addition our prototype yields more information about the design than the documentation does. The file ranking feature was very useful in several cases.

5.2 Case Study: SD/MMC Controller

The design of a controller chip for SD/MMC cards for up to 2GB, is used in this case study. The controller is accessed through a Wishbone-slave-interface. The test bench of the controller includes a Wishbone simulator and an SD-card simulator used for testing. The test bench defines five different features:

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

The test bench consists of one test case, but clearly defines when which feature is used, such that we used this distinction to measure the different coverages for the corresponding executions.
The documentation of this design 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, in the SD/MMC Controller all features are equally easy to find. They are less easy to spot than the arithmetic operations in the first case study, but far 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 *indispensable*, respectively. This practically rendered the categorization schemes useless. This is very similar to the effect which motivated us to introduce Cat_{ext}. However, this causes no problem for the schemes with continuous range (*Tarantula* and *Ochiai*) as the computed likelihood is only minimally affected, both schemes showed good results, with no

Table 4: The file ranking for the SD/MMC Controller design compared to the documentation.

Feature	*Tarantula* scheme	documentation	
	Ranking	Belongs	Possible
Register access	ctrlStsRegBl[2]		
SD init	initSD spiTxRxData spiCtrl	initSD spiCtrl	sendCMD
SD read	readWriteSDBlock spiMasterWishBoneBl[2] spiCtrl sm_RxFifoBl[1]	readWriteSDBlock spiCtrl	sendCMD
SD write	readWriteSDBlock sm_TxFifoBl[1] spiMasterWishBoneBl[2] spiCtrl	readWriteSDBlock spiCtrl	sendCMD
SPI bus access	spiCtrl spiTxRxData ctrlStsRegBl[2] readWriteSPIWireData	readWriteSPIWireData	spiCtrl spiTxRxData

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

visible differences between each other. As there are no features which are clearly *orthogonal* the comparison function of our technique was not used.

Again, after we finished our inspection we checked the documentation to find out where which feature was 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, the design is accessed through a Wishbone-interface. The Wishbone-interface identifies the commands and forwards them to the corresponding modules. Therefore, we consider the files related to the Wishbone-interface as commonly used by all features. Table 4 compares our findings with the claims of the documentation. The files are ordered based on their ranking. If the documentation clearly relates a file to a feature this file is listed in column *Belongs* and those files where the documentation is unclear, listed in column *Possible*. In case of `Register access` only the files in the ranking with a threshold of 1.0 are shown. In all other cases all files are shown until the point where all files from *Belongs* are included. Over all features only one file is included by our approach which the documentation relates to another functional behavior.

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. Our prototype supports statement coverage and toggle coverage. Four different coloring schemes to present the results have been implemented. Two 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. We also introduced a heuristic to rank file based on their likelihood of being related to a feature, allowing to guide the user faster to the corresponding code. Additionally we introduced a comparison for *orthogonal* features to improve design understanding.

The case studies emphasize the strength of our approach. They also showed that the *Tarantula* 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 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. Altogether our approach often yields more information about the implementation of the features than the documentation, even in difficult cases.

7. REFERENCES

[1] R. Abreu, 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.

[2] R. Abreu, 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.

[3] E. Clarke, M. Fujita, S. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. In *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 72–72. 1999.

[4] A. DeOrio, A. Bauserman, V. Bertacco, and B. Isaksen. Inferno: Streamlining verification with inferred semantics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(5):728 –741, 2009.

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

[6] A. Fantozzi. Locating Features in Vim: A Software Reconnaissance Case Study. Technical report, 2002.

[7] G. Fey and R. Drechsler. Improving simulation-based verification by means of formal methods. In *Asia and South Pacific Design Automation Conference*, pages 640–643, 2004.

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

[9] ITRS Working Group. International technology roadmap for semiconductors 2009 update system drivers, 2009.

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

[11] W. Li, A. Forin, and S. A. Seshia. Scalable specification mining for verification and diagnosis. In *Design Automation Conference*, pages 755 –760, 2010.

[12] R. Santelices, 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.

[13] A. Sinha, P. Dasgupta, B. Pal, S. Das, P. Basu, and P. P. Chakrabarti. Design intent coverage revisited. *ACM Transactions on Design Automation of Electronic Systems*, 14:9:1–9:32, 2009.

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

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

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