Mutation based Feature Localization

Jan Malburg*
*Institute of Computer Science
University of Bremen
28359 Bremen, Germany
malburg@informatik.uni-bremen.de

Emmanuelle Encrenaz-Tiphene†
†Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie
75252 Paris, France
Emmanuelle.Encrenaz@lip6.fr

Goerschwin Fey*‡
‡Institute of Space Systems
German Aerospace Center
28359 Bremen, Germany
Goerschwin.Fey@dlr.de

***Abstract*—The complexity of modern chip designs is rapidly increasing. More and more blocks from old designs are reused and third party IP is licensed to fulfill strict time-to-market constraints. Often, poor documentation of such blocks makes improvements and extensions of the blocks a difficult time consuming task. In this paper we present a technique for automatically localizing the parts of the code which are relevant for a feature. With this a developer can better understand the design and, consequently, can adjust the design more efficiently. The presented approach uses mutants changing the code of the design at a certain location. The code changed by a mutant is considered to be related to a feature if the mutant is killed while the feature is used. The use cases are generated using an automatic approach. This approach is based on a description specifying how the different features are used. Compared to two previous approaches the manual work is significantly reduced and the localization is of similar or even better quality.**

I. Introduction

Modern chip designs, especially *System on Chip* (SoC) designs, grow with respect to their transistor count as well as their functionality. In order to be able to fulfill strict time-to-market constraints more design blocks from previous designs are reused or third party IP blocks are licensed [1]. All those blocks provide some features for the design. Following the definition of the IEEE Standard 829 [2], a *feature* is a distinguishing characteristic of the design. A feature is typically defined with respect to functionality, robustness, or performance. In this paper we are especially interested in functional features. If extensions or improvements are done or bugs are fixed, this often relates to a set of those features. Normally, a developer starts by identifying the parts of the design relevant for the corresponding features. However, doing so can be a tedious task, especially if the corresponding code is some third party IP, some poorly documented legacy code or simply because the developer is new in the team and inexperienced with the design. In this work we propose an approach to automatically localize the code which is relevant for a feature using mutants of the design. A *mutant* is a copy of the design, which differs from the original design at one single point.

The basic assumption underlying the proposed approach is that if parts of the code are related to a feature, changes to that code may have an effect on the feature. Based on a conceptual finite state automaton of the design a set of use cases is created. During the generation of the use cases the system stores when each use case uses which feature. Those use cases are used for mutation testing. Based on the results of the mutation testing a mapping between source code and features is generated. In this work we are considering designs, for which effects of using features can be observed at the primary outputs of the design. We are not considering designs for which the features only affect the internal state of the design.

Typically, feature localization uses a set of predefined use cases, for which the features they are using are known [3]. Those use cases then are executed and coverage information is gathered. In the last step a heuristic is applied to the coverage information to find the code which is relevant for the different features. When using feature localization, the quality of the result depends on the use cases available for the computation. First, a good coverage of the design under consideration is required. Second, classical feature localization considers use cases as a whole, therefore each use case should use as few features as possible. In this work we address both problems by automatically generating use cases. The generation process is based on a description, provided by the user, specifying how the different features are used. Our automatic use case generation has two advantages. First, automation allows to create many different use cases, needed to reach high coverage. Second, for each of the use cases we do not only know which features they are using but also when they use which feature. Thus, we can lift the requirement of use cases using as few as possible features.

The contributions of a this paper are:

- Mutation testing to conduct feature localization.
- A description format for specifying how the features of a design are used.

The remainder of this paper is organized as follows: Section II presents related work. Section III introduces terms and definitions. The approach is presented in Section IV. Section V describes the proposed description of how features are used and Section VI presents the generation of use cases and optimizations during their execution. Section VII evaluates the approach and Section VIII concludes the paper.

II. Related Work

Feature localization was first proposed for software systems [4]. Feature localization compares the coverage data of runs which use the feature under consideration, with the coverage data of runs which do not use that feature. A heuristic is applied in order to decide which parts of the code are implementing the feature. In previous work we implemented feature localization for hardware designs using statement- and toggle-coverage [5]. Further, we implemented feature localization for hardware design, using dynamic data- and control-flow analysis [6]. Compared to those approaches, the approach proposed in this work has several advantages: All previous approaches consider complete use cases, which must be defined by the developer. In contrast, the proposed approach uses automatically generated use cases, targeting specific features. This allows a feedback between the localization and the use case generation process to improve the localization for parts of the code, where information is missing. Further, the proposed approach considers each use of a feature within a run independently, instead of the run as a whole.

Michael, Grosse and Drechsler proposed feature localization for *Electronic System Level* (ESL) models [7]. The models they are considering are written in SystemC, a class library for C++. This allows them to use the standard feature localization approaches for software languages based on line coverage. In contrast, we are considering feature localization for HDL-designs. Further, they are also only considering runs as a whole.

Another approach for reducing the code a developer has to consider is program slicing [8]. We distinguish between static program slicing [9] and dynamic program slicing [10]. For program slicing

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

one position in the system's code is considered, called the *slicing criterion*. Then program slicing computes those parts of the code which are affected (forward program slicing) or affects (backward program slicing) the slicing criterion. In case of static program slicing this computation is done with respect to all possible use cases and in case of dynamic program slicing with respect to one single use case. Program slicing considers the relation of parts of the code, with respect to the slicing criterion, in contrast feature localization considers the relation of parts of the code to a feature.

Mutation testing [11] is an approach for measuring and improving the quality of a test suite. For this several syntactically correct versions of the design are generated, called *mutants*. Each mutant differs from the original design at one single point in the source code. Then the test suite is applied to this mutants and it is checked if the test suite identifies the mutant as incorrect. The quality of the test suite is defined as the ratio between killed and not killed mutants. The technique presented here also generates mutants, but utilizes the mutants in order to decide which feature uses which part of the code.

In [12] a coverage-driven layered testbench architecture for the generation of randomized test cases is described. The *Standard Universal Verification Methodology* (UVM) [13] describes a set of classes helping to write a coverage-driven layered testbench architecture as well as defining best practices for simulation based verification. In this paper however, we propose an approach for feature localization, for which the use case generation is only a part of the approach. Further, coverage-driven layered testbench architectures target (functional) coverage, which not necessarily relates to features. Additionally, even if the functional coverage is defined in such a way that it relates to features, only after execution it is known which features were executed, but still without the knowledge when which feature has been used. In contrast, our approach creates use cases with specific features to use. This enables our approach to know all used features in advance and even know exactly when which feature is used. However, we believe that an existing coverage-driven layered testbench architecture can greatly help in writing a feature description as needed by our approach.

III. Preliminaries

Let H be the hardware design under consideration. We consider the initial state as part of the design. A *use case* $u = (i_0, i_1, i_2, ..., i_m)$ of H is given by a sequence of assignments $i_k, k \in \{1, 2, 3, ..., m\}$ to the primary inputs of H. We denote by $o^H[u]$ the output sequence produced by H on input sequence u.

A feature f defines the behavior of the design for a set of input sequences S_f. Let $F = \{f_1, f_2, f_3, ...\}$ be the set of all features of H. A feature relates to some part of the design's specification. A use case u *uses* a feature f if there exists an input sequence $s_f \in S_f$ such that s_f is a subsequence of u.

We call a set of two or more features *mutually exclusive*, if those features cannot be used together at the same time. Such mutually exclusiveness typically exists between features which require access to the same resources. Such resources can, e.g., be some computation unit or primary inputs for which each valuation results in another feature to be used. However, using mutually exclusive features in sequential order is allowed.

Another relation between features is their orthogonality. Given several sets of features $F_1, F_2, ..., F_n$ with:

1) $\forall x, y \in [1...n], x \neq y \Rightarrow F_x \cap F_y \equiv \emptyset$
2) $\forall x \in [1...n], (|F_x| \equiv 1) \vee (F_x$ *is mutually exclusive*$)$
3) $\forall u \in U, \forall x \in [1...n], (\exists f \in F_x, u$ *uses* $f)$
   $\Rightarrow (\forall y \in ([1...n] \backslash x), \exists f' \in F_y, u$ *uses* $f')$

We say the features in F_x are *orthogonal* to any feature in the sets $F_1, F_2, ..., F_{x-1}, F_{x+1}, ..., F_n$. Further, the sets $F_1, F_2, ..., F_n$ are pairwise orthogonal.

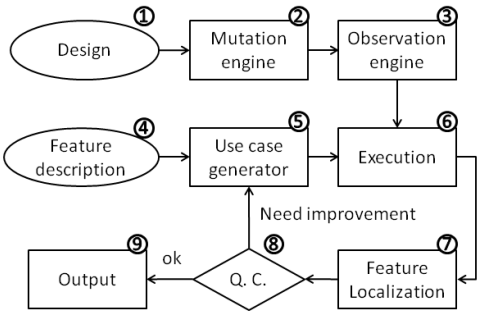

Figure 1: Overview of our approach

Informally, the sets $F_1, F_2, ..., F_n$ are pairwise disjoint and each of the sets either includes only one feature or the features they are including are mutually exclusive. If a user wants to use any of those features, he has to choose one feature from each of those sets. Such a case often appears in pipelined designs where one functionality from a set of functions can be chosen at the different stages of the pipeline.

A *mutation operator* is a function, which takes a design as input and returns another design which differs from the input design at one single chosen location in the source code. The output of the mutation operator is called a *mutant* of the original design. The change applied by the mutation operator is syntactically valid, meaning if the input design is syntactically correct then the resulting mutant is syntactically correct as well.

A mutant M is *killed* by a use case u, if $o^M[u] \neq o^H[u]$, i.e., when applying u, the values of the primary outputs of the mutant differ from the values of the primary outputs of the original design.

IV. Approach

In this section we describe the proposed technique to localize features. A basic assumption of our approach is that, if some parts of the code are related to a feature, mutants of those parts are likely to be killed by use cases which use that feature. For our approach we automatically generate use cases of the design under consideration. For these use cases we know which features of the design are currently used by the use cases. We execute those use cases on mutants of the design. If a use case kills a mutant, we relate the code which has been changed by the mutant to the feature currently used by the use case.

Figure 1 shows an overview of our proposed approach. The ovals denote input from the user of our approach. Rectangles and diamonds are automatic steps of our approach. These inputs and steps are described in the following:

1) Design: The design under consideration is given in HDL-code. For this work we are considering designs for which the effects of their features are observable at their primary outputs.

2) Mutation engine: Mutation operators are applied to the design in order to create mutants. Following mutation operators are used by our approach:

- Operand mutation: An operand of an expression is either replaced by the constants where all bits are one, by the constant zero, or by its negation.
- Operator mutation: The operator of an expression is replaced by another operator of the same type, e.g. a bitshift-operator is only replaced by another bitshift-operator.
- Condition negation mutation: The condition of a loop, an if-condition, an event expression, or a switch-statement is negated. In case of a switch-statement a bit-wise negation is used, otherwise a logical negation.
- Assignment mutation: The right-hand-side of an assignment is replaced by its bit-wise negation, the constant zero, or the constant for which all bits are set to one.

- Basic block removal mutation: An always statement, initial statement, an loop, or an conditional statement is removed including all statements it is guarding.

3) Observation engine: For each position in the source code, an observation module is generated. This observation module instantiates the original design and all mutants of the corresponding source code position. The observation module has the same interface as the original design, broadcasts all its inputs to its sub-modules and outputs the outputs of the original design, thus it can be used as replacement of the original design.

4) Feature description: A description, created by the user of our technique, specifies which features are supported by the design, how those features are used, and under which conditions they can be used. The description is given as an automaton capturing the constraints for the activation of features. To describe how the different features are used SystemVerilog tasks are utilized. For detailed information see Section V.

5) Use case generator: Several use cases are created, using the different features of the design. For the generated use cases it is known when they use which feature. For detailed information see Section VI.

6) Execution: The use cases are applied to the different observation modules and it is recorded which features were used while an observer module detects a mismatch of the outputs of the original design and a mutant. For information about the optimization we are using during this step see Section VI.

7) Feature localization: The feature localization maps those parts of the code to the features which were executed while a mismatch between the corresponding mutants and the original design has been detected.

8) Quality Check: The result of the feature localization is rated regarding the quality and rules are used, to decide whether further use cases are needed and which features should be used. The current implementation uses following rules whether to stop the creation of new use cases:

- The creation stops if either 95% of all observers have been killed by at least one use case,
- or a user defined number of use cases is reached.

When creating a use case the first applicable of the following rules are respected:

- Use a feature not used by any use case.
- Use a feature not having killed any mutants yet.
- Use a feature which is used less times than half the average number of how often the other are used.
- Create a random use case.

9) Output: The computed result of the feature localization is presented to the user.

V. Feature description

We propose to model the feature interactions and activations with an automaton on an abstract level.

Let $F = \{f_1, ...f_n\}$ be the set of features; we introduce the abstract automaton $\mathcal{A} = (S, S_0, 2^F, \delta)$ with $\delta \subseteq S \times 2^F \times S$ where 2^F denotes the powerset of F. The states in S describe the abstract states of the system where particular features can be activated. Transitions represent the execution of a subset of features.

Example 1. *Figure 2 shows an example automaton with two states and the features Reset and Check. If a Check fails, the design enters an error state and a Reset must be applied.*

The automaton $\mathcal{A}$ is a specification of features and their interaction. We manually generate a refined automaton $\mathcal{B} = (S, S_0, A, \delta')$ with $\delta' \subseteq S \times A \times S$. The states S and the starting state S_0 are identical with those from $\mathcal{A}$. Instead of sets of features $\mathcal{B}$ uses a set A of

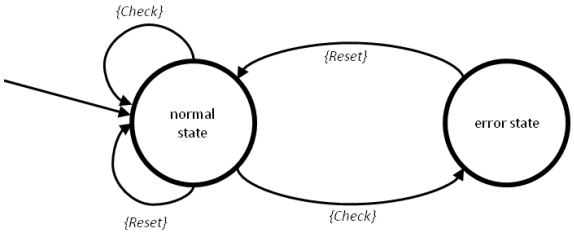

Figure 2: The abstract automaton for our example

input sequences as alphabet. Those input sequences are represented by SystemVerilog tasks. When refining $\mathcal{A}$ into $\mathcal{B}$ a set of features is replaced by one or more SystemVerilog tasks, which together use exactly that set of features. We do not allow non-determinism in $\mathcal{B}$. Two transitions with the same set of features may be refined using different tasks in order to remove non-determinism. We are using an implicit representation of the automaton $\mathcal{B}$. The automaton $\mathcal{B}$ is described using SystemVerilog tasks with additional keywords defining the transitions of the automaton. The description format is presented in the following.

A. Basic description elements

Essentially, the description encodes the transitions of $\mathcal{B}$ as *actions*. The simplest type of actions are SystemVerilog tasks describing stimuli of the design annotated with additional information, e.g. the list of features they are using. This simple type of actions is labeled by the keyword `#action`. We call those *simple actions*. New keywords are introduced to capture the behavior of $\mathcal{B}$. Each simple action contains at least three sections. First, a unique SystemVerilog task describes the input stimuli. The parameters of the task can be restricted with the `@range` annotation. Second, `@feature` lists the used features' names. Finally, a `@ready` section detects the completion of the action.

If the automaton contains more than one state, these states are described using state variables. State variables are defined using the `#states` keyword. An action may have a `@req` section. The `@req` section describes requirements over the state variables, which must be fulfilled in a state for the action to be allowed to be used in that state. We say an action t is *enabled* in a state $s \in S$ if s fulfills all requirements of t. A simple action describes as many transitions as there are states, which fulfill its requirements. An `@effect` section describes the state change caused by the action. If the `@req` section is missing, the action can be used in any state. If the `@effect` section is missing, the action describes self-transitions.

A special type of action is the initialization action. This action is described in the `#init` section. The initialization action is the first action executed whenever the design is used and is never considered to be enabled in any state. The initialization action may include a SystemVerilog task, but this task must not have any parameters. If state variables are defined the `#init` section is mandatory and its `@effect` annotation must uniquely describe the starting state of the automaton.

Example 2. *In Listing 1 we see an excerpt of the description for a controller implementing a communication protocol. The protocol requires an initial handshake. The features of this design are:* connect*,* send*, and* disconnect*. In this case, the* send *feature can only be used if the design is currently connected. During the initial handshake the data-rate of the connection will be negotiated. We assume that only rates between 10,000 and 100,000 bytes per second are allowed. Thus, we have a restriction of the parameter for the data-rate.*

B. Advanced Description Elements

Simple actions are only able to encode at most one transition per state in $\mathcal{B}$. Additional modeling elements allow for a more

```
#states: connected

#features: connect, send, disconnect

#init:
        task init();
        rst=1;
        #2 rst=0;
        endtask;
        @effect: connected=0

#action:
        @feature: connect
        @req: !connected
        @range: 10000 <= bps <=100000
        task connect(integer bps);
        ...
        endtask
        @effect: connected=1
        @ready: connect_interrupt==1
        ...
```

Listing 1: Example definition of features.

compact description. The additional description elements specify the transitions for all combinations of orthogonal features[1] in linear space. The constrained random simulation creates single transitions from those constructs whenever required during the generation of use cases.

For describing orthogonal features we are using *orthogonal actions*. The idea for describing orthogonal features is that for each feature a SystemVerilog task is defined. Additional rules describe how to combine those tasks into a transition. We call those SystemVerilog tasks *partial actions*. For describing orthogonal features we use three additional types of sections: `#partial`, `#group`, and `#orthogonal`. The type `#partial` defines a partial action and optionally contains a list of features which the partial action uses. We explicitly allow `#partial` sections without features, for example to set primary inputs which do not decide the features, but rather set the operands. A `#partial` section may further contain a `@ready` annotation. The `#group` type contains a list of partial actions. Basically, a `#group` describes a group of mutually exclusive features. Each `#partial` section and `#group` section has a unique name to reference it. The `#orthogonal` section defines an orthogonal action and describes how the different partial actions and groups can be combined into transitions. Additionally, the `#orthogonal` section, like simple actions, may define requirements and effects over the state variables. For describing the different combinations of partial actions we use the following set of basic combination operators:

- Parallel (|): The tasks start concurrently.
- Sequential (>): Each task starts after the previous task in the defined order has finished.
- Unordered (^): Like sequential but using a random permutation as order.

For describing the combination we are using following syntax:

```
FORMULA = "("OPERAND"|"OPERAND{"|"OPERAND}")"|
          "("OPERAND">"OPERAND{">"OPERAND}")"|
          "("OPERAND"^"OPERAND{"^"OPERAND}")"
OPERAND = FORMULA | unique_name
```

where `unique_name` means the name of a partial action or group. The use of a group is interpreted such that a single randomly chosen element of that group is used. In case of parallel tasks, the user is responsible to ensure, that they do not write concurrently to the same signal.

Example 3. *In Listing 2 we see an excerpt from the description of orthogonal features. Excerpts of each section type are shown.*

[1]which is an exponential large amount of actions in the number of orthogonal feature sets

```
...
#partial
        @name: start
        task start(bit[0:63] opa, bit[0:63]opb)
        ...
        endtask
        @ready:#1
...

#group
        @name:a_op
        @list: add,sub
...

#orthogonal
        @req:error=0
        @formula ((a_op ^ r_mode)>start)
        @ready:ready==1
```

Listing 2: Example showing definition of orthogonal features.

VI. Use Case Generation and execution

Each valid sequence of transitions is a use case of the design. First the description is used to compute the automaton $\mathcal{B}$. For this a simple graph search algorithm, generating new states and transitions on the fly, is used. Starting from the initial state, in each reached state the set of enabled actions is computed. For each enabled action a transition, annotated with that action, is created. This may create new states which are checked later as well.

In the next step we compute which feature can be used from which state. For this we map each feature f to a subset $S'_f \subseteq S$ of the states. A state $s \in S$ is included in S'_f if and only if there exists an action t such that t uses f and t is enabled in s.

The automaton $\mathcal{B}$ and the mapping are utilized to create a use case using a wanted feature f as follows: First, we choose a state $s \in S'_f$ which is mapped to f. Then, we compute a sequence of actions $p = (t_0 > t_1 > ... > t_n), \forall x \in [0...n], t_x \in \Sigma$ which ends in the state s when applied to the starting state. To this path we append an action t_f which is enabled in s and uses f. For each action $t \in \{t_0, ..., t_n, t_f\}$, if t is an orthogonal action, we create a random sequence of its partial actions fulfilling the combination rule. If t_f is an orthogonal action we have to take care, that the resulting combination of partial actions uses f. In the last step random parameters are chosen for each task in each action of $\{t_0, ..., t_n, t_f\}$. This results in a use case $U_p = (I > t_0 > t_1 > ... > t_n > t_f)$, where I denotes the initialization action. A random use case is created by computing a random path through $\mathcal{B}$.

The resulting use case is in principle a concatenation of calls to the corresponding tasks. Additional code to wait for the ready conditions and ensuring the combination rules for orthogonal actions is added.

While executing the use case on the different observer modules a set of optimizations are used. This contains four optimizations. The first optimization is commonly used for mutation testing and the other three are new optimizations fitted to our approach:

1) Only execute a mutant with use cases covering the mutated statement.
2) Do not execute mutants killed by the initialization action for any further use case.
3) Do not execute mutants which are not killed by mutating the control statement guarding the mutant.
4) Stop the execution as soon as a difference at the primary outputs is observed.

VII. Evaluation

In this section we evaluate our approach. For this evaluation we compare the presented approach against previous approaches for feature localization based on coverage metrics [5] and feature localization based on dynamic dataflow analysis [6]. The first approach uses statement- and toggle-coverage of different runs and applies

Table I: Comparison of the manual effort and the utilized use cases for the different approaches.

Approach based on	Manual effort	# of use cases	# of operations
Coverage	17,924 LoC + mapping use cases to features	320	320
Dynamic dataflow	17,924 LoC + mapping use cases to features + marking of primary inputs/outputs	320	320
Mutation	96 LoC	128	265

a heuristic to decide which statements and signals are related to a features. The approach based on dynamic dataflow analysis computes the data-path from the primary inputs, which decide the features to be used to the primary outputs which provide the result of the features. All parts of the source code on this path are considered to be covered. Based on our previous experiments [5] we are using an adaption of the Tarantula-heuristic [14] as heuristic for those approaches. This heuristic uses the percentages of how often a part of the design was covered while the feature was used and the percentage of how often that part was covered while the feature was not used. The result is a likelihood whether the code belongs to a feature and a confidence value describing the probability that the likelihood is correct.

For the coverage based and dynamic dataflow based approaches the manual work is considerably more time consuming than for the approach presented in this paper. Both before mentioned approaches require that a user writes use cases for the design and maps them to the features. For the approach using dynamic dataflow analysis the user additionally must mark whether primary inputs and primary outputs are related to the feature or not. If the primary input and primary outputs are not related to a feature all the time, this marking must be done for each clock cycle which is especially tedious.

A. Design for evaluation

For the evaluation we use a Floating-Point-Unit (FPU) from the OpenCores website[2]. This design consists of 1,710 *Lines of Code* (LoC) in seven different modules, where each module is contained in its own file. This design has eight features: the arithmetic operations *addition*, *subtraction*, *division*, and *multiplication*; and the rounding modes *round to zero*, *round to nearest even*, *round to -infinite*, and *round to +infinite*. The set of rounding modes and the set of arithmetic operations are orthogonal to each other.

B. Manual effort and use cases

Table I compares the required manual effort for using the different approaches as well as the set of use cases used for creating the localization. In the column *Manual effort* we show which work a user had to do in order to use an approach. The manual effort for writing the use cases or the feature description, respectively, is given as non-empty, non-comment LoC. The column *# of use cases* gives the number of use cases utilized for each approach and column *# of operations* gives the number of operations executed by the use cases. As operation we count any action which uses at least one feature of the design. Reset and idle cycles are not counted as operations.

With the proposed approach, writing the feature description took about ten minutes. Beside the orthogonal action describing all features of the design, the description contains two simple actions one for reset and one creating an idle clock cycle. The initialization action resets the design. The following parameters are used for our approach: a use case limit of 128 for the quality check step and a maximum length of 8 transitions for the use case generation step.

The other approaches require user defined use cases annotated with the features they are using. For those approaches we use a set of 20 different pairs of operand values. Each pair is applied to each combination of arithmetic operation and rounding mode resulting in 320 use cases. For the approach using dynamic dataflow analysis we marked the primary inputs which decide the arithmetic operation and rounding mode as start of the data path. As the end of the data path we use all primary outputs of the design at the clock cycle where the design indicates the end of the operation.

[2]http://opencores.com/project,double_fpu

Table II: Ranking of the files for the different approaches

Feature	documentation	Coverage	Dynamic dataflow	Mutation
addition	fpu_add fpu_sub fpu_double	fpu_sub fpu_exceptions fpu_double fpu_add	fpu_double fpu_sub fpu_add	fpu_add fpu_double fpu_sub
subtraction	fpu_add fpu_sub fpu_double	fpu_add fpu_double fpu_exceptions fpu_sub	fpu_double fpu_sub fpu_add	fpu_double fpu_add fpu_sub
division	fpu_div	fpu_div	fpu_div	fpu_div
multiplication	fpu_mul	fpu_mul	fpu_mul	fpu_mul
round to zero	fpu_round fpu_exceptions	fpu_round fpu_exceptions	fpu_round fpu_exceptions	fpu_exceptions fpu_sub fpu_double fpu_round
round to nearest even	fpu_round fpu_exceptions	fpu_round fpu_exceptions	fpu_exceptions fpu_round	fpu_add fpu_exceptions fpu_div fpu_round
round to -infinite	fpu_round fpu_exceptions	fpu_round fpu_exceptions	fpu_exceptions fpu_round	fpu_round fpu_exceptions
round to +infinite	fpu_round fpu_exceptions	fpu_round fpu_exceptions	fpu_exceptions fpu_round	fpu_round fpu_exceptions

C. File ranking

All three approaches for feature localization allow to rank the files of the design with respect to the likelihood whether they contain code relevant for a feature. The documentation gives the relation between features and code at file level. First, we compare the computed ranking with the documented relation.

Table II shows the file ranking for the different approaches. The ranking is shown until all files are included, which the documentation relates to an feature..

For the *addition* and the *subtraction* feature, based on the sign of the operands, the operation may be executed by the *fpu_sub* module or the *fpu_add* module. The top module is contained in *fpu_double* and is responsible for this decision. The file ranking prioritizes high likelihood values. Therefore it is affected by false positives. This effect can be seen for the coverage based approach with the *addition* and the *subtraction* feature as well as for the mutation based approach with the features *round to nearest even* and *round to zero*.

Based on the file ranking the approach using dynamic dataflow analysis results in the best localization, with an optimal result for all features. With six optimal localizations each, the coverage metric based approach and the presented approach reach equally good results.

D. Expression level

Next, we will compare the results on statement and expression level. For this we chose two representative features and have an in-depth comparison of the computed feature localization for these features. The results for all arithmetic operations are similar to each other, the same is true for the rounding modes. Therefore, we chose *multiplication* as representative for the arithmetic modes (Table III) and *round to +infinite* as representative for the rounding modes (Table IV). We categorized the statements and expressions with respect to two dimensions. The horizontal dimension categorizes whether the localization considers them as part of the feature (part), as part of all features (all), or as not part of the feature (not part). In the vertical dimension we divide between a high confidence that the categorization is correct (high) and a low confidence (low). The tables give the percentage values for how much of the file is categorized into which class. Statements not covered and mutants not killed for any use case are placed in the class low confidence and not part

Table III: Results for the multiplication feature.

File	conf.	Coverage			Dynamic dataflow			Mutation		
		part	all	not part	part	all	not part	part	all	not part
fpu_add	high	0%	57%	0%	0%	0%	0%	0%	0%	0%
	low	0%	0%	43%	0%	0%	100%	0%	0%	100%
fpu_div	high	0%	41%	0%	0%	0%	0%	0%	0%	0%
	low	0%	0%	59%	0%	0%	100%	0%	0%	100%
fpu_double	high	12%	79%	0%	5%	17%	6%	5%	46%	2%
	low	0%	0%	9%	0%	0%	71%	0%	2%	46%
fpu_exceptions	high	8%	92%	0%	0%	11%	1%	9%	39%	0%
	low	0%	0%	0%	9%	2%	77%	4%	6%	42%
fpu_mul	high	65%	2%	0%	51%	0%	0%	49%	0%	0%
	low	7%	0%	27%	5%	0%	44%	3%	0%	48%
fpu_round	high	4%	96%	0%	0%	60%	0%	2%	45%	0%
	low	0%	0%	0%	0%	17%	23%	0%	41%	12%
fpu_sub	high	0%	30%	0%	0%	0%	0%	0%	1%	1%
	low	0%	0%	70%	0%	0%	100%	0%	1%	97%

Table IV: Results for the round to +infinite feature

File	conf.	Coverage			Dynamic dataflow			Mutation		
		part	all	not part	part	all	not part	part	all	not part
fpu_add	high	0%	57%	0%	0%	0%	0%	8%	1%	0%
	low	0%	43%	0%	0%	68%	32%	0%	58%	33%
fpu_div	high	0%	41%	0%	0%	0%	0%	0%	28%	0%
	low	0%	19%	39%	0%	47%	53%	1%	20%	51%
fpu_double	high	0%	87%	0%	0%	24%	0%	0%	58%	0%
	low	0%	13%	0%	0%	29%	48%	0%	24%	19%
fpu_exceptions	high	2%	98%	0%	0%	12%	0%	2%	41%	1%
	low	0%	0%	0%	1%	9%	78%	1%	13%	42%
fpu_mul	high	0%	35%	0%	0%	0%	0%	0%	0%	0%
	low	0%	39%	27%	0%	56%	44%	0%	50%	49%
fpu_round	high	4%	96%	0%	0%	58%	4%	6%	51%	0%
	low	0%	0%	0%	4%	15%	19%	4%	22%	16%
fpu_sub	high	0%	30%	0%	0%	0%	0%	0%	29%	1%
	low	0%	29%	41%	0%	50%	50%	0%	11%	58%

of the feature. The files which are related to the feature by the documentation are indicated by a gray table background.

Let us discuss the *multiplication* feature first. All three approaches identify the corresponding file as the primary part of the feature. Classifying code from this file as not part of the feature has two different reasons. First, the file contains a switch-case block of 54 branches, however no approach covers all branches. By default, code not covered is considered as not part of any feature. Additionally, the mutation based and dynamic dataflow based approach exclude the code responsible for the reset from the localization. The code localized in the files *fpu_round* and *fpu_exceptions* are mostly false positives, however some code in *fpu_round* is correctly localized as it creates interrupt signals in case of multiplication by zero or infinite. The file *fpu_double* is the top module of the design and multiplexes the input to the different sub-modules. Again, the approach based on coverage metrics also marks the reset code of the corresponding registers as part of the feature. Overall, the approach presented in this paper and the approach based on dynamic dataflow create equally good localizations in case of the arithmetic operation. The results using coverage metrics are less good, as code for resetting the design is included.

When considering the *round to +infinite* feature, first we see that the approach using coverage metrics considers most code as part of all features. The code considered as not being part of the feature is code which is not covered. The code which is considered as part of the feature is only considered so due to one signal in each of the two corresponding files. This signals are high exactly when the rounding mode is used. Besides the statements setting those signals four additional statements, including the reset of those signals, are marked as part of the feature. The same effect can be observed for all other rounding mode explaining the good file ranking for all rounding modes while using the coverage based approach. However, this localization provides poor help for a developer. The code found by the dynamic dataflow based approach is part of the feature. However, that approach also misses large parts of the code belonging to the feature. The mutation based approach finds the main parts of the rounding feature, marks them as part of the feature, and also correctly identifies the part that *round to +infinite* shares with the other rounding modes. However, the mutation based approach marks some code in the files *fpu_add* and *fpu_div* incorrectly as part of the feature, i.e., includes false positives. Altogether, for the rounding modes, the dynamic dataflow based approach provides an under-approximation of the code of the features, while the mutation based approach provides an over-approximation. Depending on the task of the developer the one or the other is better. Overall the mutation based approach and the dynamic dataflow based approach provide similar good localizations.

VIII. Conclusion

In this paper we presented an approach for automatic feature localization based on mutation testing and automatically generated use cases. The only manual work required by the user is writing a description of how the different features of the design are used. The system then creates mutants of the design under consideration. A set of automatically generated use cases is applied to the mutants to check the use of which feature kills which mutants. Based on this the code of the design is mapped to the different features of the design.

We compared the presented technique against two previous approaches one based on statement and toggle coverage and one based on the analysis of the dynamic dataflow. Compared to the approach based on coverage metrics, the presented approach provides the better localization result, while requiring significantly less manual work. The approach based on dynamic dataflow analysis creates similarly good localization as the presented approach, but requires most manual effort of all considered approaches.

References

[1] ITRS Working Group, *International Technology Roadmap for Semiconductors 2009 Update System Drivers*, ITRS Std., 2009.

[2] *IEEE Standard for Software and System Test Documentation*, Std for Software Test Documentation Working Group Std., 2008.

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

[4] 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.

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

[6] J. Malburg, A. Finder, and G. Fey, "Tuning dynamic data flow analysis to support design understanding," in *Proceedings of Design, Automation and Test in Europe*, 2013, pp. 1179–1184.

[7] M. Michael, D. Grosse, and R. Drechsler, "Localizing features of ESL models for design understanding," in *Forum on Specification and Design Languages*, 2012, pp. 120–125.

[8] 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*, ser. Lecture Notes in Computer Science, 1999, vol. 1703, pp. 72–72.

[9] M. Weiser, "Program slicing," in *Proceedings of International Conference on Software Engineering*, 1981, pp. 439–449.

[10] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, pp. 155–163, 1988.

[11] Y. Serrestou, V. Beroulle, and C. Robach, "Functional Verification of RTL Designs driven by Mutation Testing metrics," in *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, 2007, pp. 222–227.

[12] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale, *Verification methodology manual for SystemVerilog*. Springer, 2006.

[13] *Universal Verification Methodology (UVM) 1.2 Class Reference*, Accellera Systems Initiative Std., June 2014. [Online]. Available: http://www.accellera.org/downloads/standards/uvm/UVM_Class_Reference_Manual_1.2.pdf

[14] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for fault localization," in *Proceedings of ICSE Workshop on Software Visualization*, 2001, pp. 71–75.