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 propose 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 proposed 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.

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 and 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 of the design contains a modification at a single location of the original source code.

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. We propose applying use cases, for which we know the features they are using, to mutants of the design. If a mutant is killed, we check which feature was currently executed by the design, and the part of the design changed by the mutant is considered to be related to that feature. In this work we are considering designs, for which the result of the 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, e.g., a CPU design writing the result of the operations which are features to the registers.

When using feature localization, the quality of the result depends on the use cases available for the computation. On one hand a good coverage of the design under consideration is required; on the other hand 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. One goal is that a developer should easily be able to write such description. An automaton describes at an abstract level how features may be used. From this description a test harness is created which will be implemented as an extension of SystemVerilog. We do not require a developer to define all possible ways of how to use a feature. Instead the developer only has to specify the use of the features to an extent he considers sufficient. Our automatic use case generation has two advantages. First, automation allows to have as much different use cases as 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:

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

The remainder of this paper is organized as follows: In Section II related work is presented. Section III introduces terms and definitions used throughout the paper. The proposed approach is presented in Section IV. The model used for specifying the features' interaction is described in Section V. Section VI describes the proposed description of how features are used and how the description is utilized to create use cases. In Section VII presents the approach on a small example and Section VIII summarizes the paper.

II. Related Work

Feature localization was first proposed for software systems [3]. 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 [4]. Further, we implemented feature localization for hardware design, using dynamic data- and control-flow analysis as input for the feature localization heuristic [5]. Compared to those approaches, the approach proposed in this work has several advantages: All previous approaches consider complete use cases, which must been defined by the developer. In contrast, the proposed approach uses automatically generated use cases, targeting specific features. This further 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 [6]. 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. We distinguish between static program slicing [7] and dynamic program slicing [8]. For program slicing 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. Originally, program slicing was proposed for software system, but there are also implementations for HDL-designs [9]. Program slicing considers the relation of parts of the code, with respect to the slicing criterion, in contrast feature localization, like the proposed approach, considers the relation of parts of the code to a feature.

Mutation testing [10] 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, also called the test suite *kills* the mutant. The quality of the test suite is defined as the ratio between killed and not killed mutants. In combination with some automatic test case generation approaches, like a genetic algorithm, mutation testing can be used to improve the test suite of a design. The technique presented here also generates mutants, but utilizes the mutants in order to decide which feature uses which part of the code.

In [11] a coverage-driven layered testbench architecture is described. In such architectures the testbench consist of five layers: signal layer, command layer, functional layer, scenario layer, and testbench layer. With respect to the work described in this paper, the scenario layer and the functional layer are especially interesting. In the scenario layer, randomness is introduced and function sequences are generated, which are split into several functions in the functional layer. In this paper however, we propose an approach for feature localization, for which the use case generation is only a part of the approach. As our approach intends to only create use cases which use defined feature and not require defining correctness conditions for the result, our approach is likely easier for a developer to use. If a coverage-driven layered testbench architecture already exists for the design, this architecture most likely could be used for the mutation based feature localization presented in this paper, possibly extended by the state-based-use case generation approach presented in this paper.

III. Preliminaries

Let H be the hardware design under consideration. We consider the initial state a 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. Let U be the set of all possible use cases. We denote by O^H the set of all output sequences created by the design H for all use cases in U, and $o^H[u]$ the output sequence produces by H on input sequence u.

A feature f defines the behavior of the design for a set of input sequences I_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 $i_f \in I_f$ such that i_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 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.

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

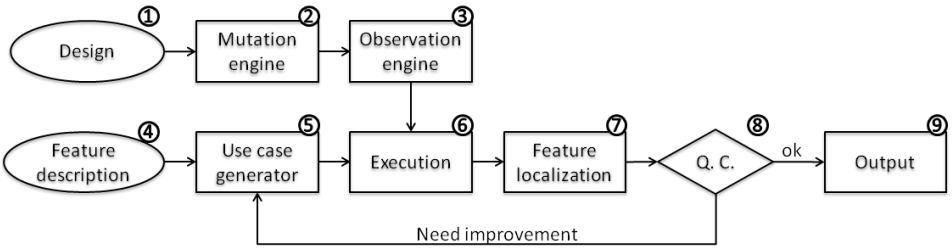

Figure 1: Overview of our approach

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 randomly chosen point. 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 as well. Some often used mutation operators are [12]:

- Constant replacement: A constant is replaced by another constant of the same type and width.
- Operand replacement: An operand is replaced by another signal or constant of the same type.
- Operator replacement: An operator is replaced by another one.
- Condition negation: A (sub-)expression of a branch-condition is negated.

A mutant M is called an *equivalent mutant*, if $\forall u \in U, o^M[u] \equiv o^H[u]$, i.e., for all use cases the values of the primary outputs of the mutant are identical to the values of the primary outputs of the original design.

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 are input from the user of our approach. Rectangles and diamonds are automatic steps of our approach. The different inputs and steps are described in the following:

1) Design: The design under consideration is given as 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. The mutation operators are restricted, such that they do not change module instantiations or the left-hand-side of assignments.

3) Observation engine: For each position in the source code, an observation module is generated which multiplexes the inputs of the original design to all mutants of the source code location. Further, an observation module compares the outputs of the mutants with the output of the original design. The outputs of the original design are used as outputs of the observation module, thus the observation module can be used like the original design. The module notifies the overall system if a mismatch is detected. Because the observation module considers several mutants for each source code position, we are assuming that it is rather unlikely that some code is related to the result of a use case without the observation module noticing this fact.

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. An extended SytemVerilog description derived from this specification serves as a test harness to generate use cases. For detailed information see Section VI.

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 a checker detects a mismatch between the original design and a mutant.

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. A localization is considered to be of poor quality if much code is not related to any feature or if for much of the code the corresponding mutants were

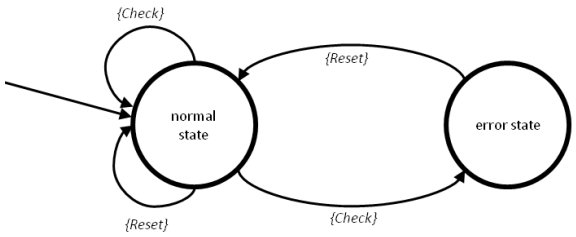

Figure 2: The abstract automaton for our example

killed before all features were applied to them. If this check decides that the feature localization is not yet good enough, the technique creates further use cases targeting those features for which the localization is not yet sufficient.

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

V. Specification of Feature Interactions

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 $\delta \subseteq S \times 2^F \times S$ and 2^F denotes the powerset of F. The states S describe the abstract state of the system where particular features can be activated. Transitions represent the execution of a subset of features. Along an abstract transition, each feature labeling the transition is activated, executes, and finishes before going to the target state. With this model, we can express, e.g., the orthogonality of features: Assume two orthogonal sets of features F_1 and F_2; the number of outgoing transitions of each state of $\mathcal{A}$ is bounded by $|F_1| \cdot |F_2|$. Figure 2 shows an example automaton with two states and the features *Reset*, and *Check*.

The automaton $\mathcal{A}$ is a specification of features and their interaction.

We generate a refined automaton $\mathcal{B}$. Instead of sets of features $\mathcal{B}$ uses 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 a SystemVerilog task, which uses 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 $\mathcal{B}$, which will be presented in the next section.

VI. State based use case generation

This section explains how the automaton $\mathcal{B}$ is described using SystemVerilog tasks with additional keywords. The automaton is used to generate a test harness. This test harness generates use cases by means of constrained random simulation.

Essentially, the description encodes the transitions of $\mathcal{B}$ as *actions*. Actions are SystemVerilog task annotated with additional information, e.g. the list of features they are using. New keywords are introduced to capture the behavior of $\mathcal{B}$. Each action is labeled by the keyword `#action`. Each action contains at least three sections. First, `@feature` lists the used features' names. Second, a SystemVerilog task describes the input stimuli. Finally, a `@ready` section detects the completion of the action. An action might further have a `@req` section constraining the state in which the action can be used and an `@effect` section describing 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.

The starting state and a task to initialize the design can be defined in an `#init` section.

```
1  #states: connected
2
3  #features: connect, send, disconnect
4
5  #init:
6    task init();
7    rst=1;
8    #2 rst=0;
9    endtask;
10   @effect: connected=0
11
12 #action:
13   @feature: connect
14   @req: !connected
15   @range: 10000 <= bps <=100000
16   task connect(integer bps);
17   ...
18   endtask
19   @effect: connected=1;
20   ...
21
22 #action
23   @feature: send
24   @req: connected
25   task send(bit[8] data);
26   ...
27   if( ERROR )
28     connected=0;
29   ...
30   endtask
31   @ready: send_interrupt=1;
32   ...
```

Listing 1: Example definition of features.

Example 1. *We are considering a controller for 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.*

In Listing 1 we see an excerpt of the description, specifying the use of those features. We describe the state of the design with the help of Boolean variables. In the `#states` *section the used state variables are defined (line 1). When state variables are used, an initialization (*`#init`*) section is mandatory (lines 5-10). This section defines a task, which is used for the initialization of the design, for example by resetting the design (lines 6-9). The initialization section includes an* `@effect` *annotation (line 10), which defines the value of the state variables after the initialization task is executed.*

An action can be annotated with a requirement (`@req`*), consisting of a constraint over the state variables, which must be fulfilled in order to be able to use that action (line 14 and line 24). Actions can change the current state of the automaton (line 19). Actions may set any subset of the state*

```
1  module parity(clk, rst, in,
2    done, error);
3    input wire clk, in, rst;
4    output reg done, error;
5    reg parity;
6    reg[0:3] pos;
7
8    initial
9    begin
10       done<=0;
11       error<=0;
12       parity<=0;
13       pos<=0;
14   end
15
16   always @(posedge clk)
17   begin
18     if(rst == 1)
19     begin
20       done<=0;
21       error<=0;
22       parity<=0;
23       pos<=0;
24     end
25     else if(error == 1)
26     begin
27       //do nothing
28     end
29     else if( pos <9 )
30     begin
31       parity <= parity ^ in;
32       pos <= pos+1;
33       done<=0;
34     end
35     else
36     begin
37       error<= parity != in;
38       done <= 1;
39       pos <=0;
40     end
41   end
42 endmodule
```

(a) Design

```
1  #features: Check, Reset
2  #states: error
3
4  #init
5    task init();
6    startClk();
7    endtask;
8    @effect:error=0;
9
10 #action
11   @feature:Reset
12   task reset();
13   rst=1;
14   #2 rst=0;
15   endtask;
16   @ready:#0;
17   @effect:error=0;
18
19 #action
20   @feature: Check
21   @req:error=0;
22   task correct(bit[0:7] v);
23   for(int i=0;i<8;i++)
24   begin
25     in=v[i]; #2;
26   end
27   in=^v;
28   endtask;
29   @ready:done=1
30
31 #action
32   @feature: Check
33   @req:error=0;
34   task incorrect(bit[0:7] v);
35   for(int i=0;i<8;i++)
36   begin
37     in=v[i]; #2;
38   end
39   in=!(^v);
40   endtask;
41   @ready:done=1
42   @effect:error=1;
```

(b) Description

Figure 3: Code of our example design

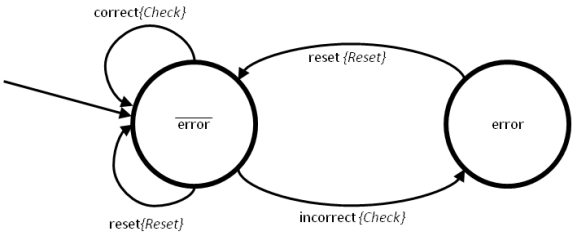

Figure 4: The concrete automaton for our example

variables. Exceptional changes to the state variables can be defined within the task (line 28). Parameters of a task can be restricted with the `@range` *annotation (line 15).*

This description requires to encode each action of $\mathcal{B}$. Additional modeling elements allow for a more compact description. This elements allow to describe sets of actions which only differ in the order they apply inputs to the design and to describe all actions 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.

VII. Applying Mutation for Feature Localization

In this section we illustrate our approach of using mutation for feature localization on a simple design. The example design is shown in Figure 3a and the corresponding feature activation description in Figure 3b. The concrete automaton $\mathcal{B}$ for this example is shown in Figure 4. The features used by the actions are shown in braces. The design checks if the correct parity bit is appended after each packet of eight bits. If so, the design raises a done signal for one clock tick and then begins with the next set of inputs. If a parity error is detected, the design raises an error signal and stays in an error state until the design is reset. Our approach starts with creating mutants of the design. For describing our approach we will consider the following mutants: M_1: line 21 '0' replaced by '1'. M_2: line 29 '<' replaced by '!='. M_3: line 29 '<' replaced by '>='.

In the next step observer modules for the mutants are created. In this step the mutants M_2 and M_3 are combined in a single observer module as they are mutants of the same source code position.

In the next step use cases are created. Let the created uses cases be: u_1: (reset, correct(246)), u_2: (correct(72), incorrect(19), reset). When applied to the observer modules M_1 is killed by both use cases while reset is used. M_2 is not killed for any use case, however the observer module includes M_3 which is killed for both use cases whenever the check feature is used.

In the next step the feature localization is applied, resulting in line 21 is related to the reset feature and line 29 to the check feature. The classification passes the quality check as both considered lines are related to a feature and further both features were applied to both observer modules.

VIII. Summary and Outlook

In this paper we proposed an approach for feature localization in hardware designs, using mutants. The basic idea of our approach is that if a mutated statement changes the behavior of a features, this statement is part of the code implementing the feature. Our approach uses the source code of the design, and a description how to use the different features to automatically perform the localization. Using the description, the required use cases are generated automatically. The description considers the design as a finite state automata, where transitions are done using the features of the design. The transitions themselves are described in form of annotated SystemVerilog-tasks. In the next step we want to extend our approach for designs, for which the result of a feature is not observable at its primary outputs, e.g., a CPU design for which the result of the computation is stored in its general purpose registers.

[1] which is an exponential large amount of actions

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 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.

[4] 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.

[5] 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.

[6] 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.
[7] M. Weiser, "Program slicing," in *Proceedings of International Conference on Software Engineering*, 1981, pp. 439–449.
[8] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, pp. 155–163, 1988.
[9] 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.
[10] 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.
[11] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale, *Verification methodology manual for SystemVerilog*. Springer, 2006.
[12] G. Al-Hayek and C. Robach, "From design validation to hardware testing: A unified approach," *Journal of Electronic Testing*, vol. 14, no. 1-2, pp. 133–140, 1999.