Tuning Dynamic Data Flow Analysis to Support Design Understanding

Jan Malburg*   Alexander Finder*   Görschwin Fey*†

*University of Bremen, 28359 Bremen, Germany
{malburg, final, fey}@informatik.uni-bremen.de
†German Aerospace Center, 28359 Bremen, Germany
Goerschwin.Fey@dlr.de

Abstract—**Modern chip designs are getting more and more complex. To fulfill tight time-to-market constraints, third-party blocks and parts from previous designs are reused. However, these are often poorly documented, making it hard for a designer to understand the code. Therefore, automatic approaches are required which extract information about the design and support developers in understanding the design.**

In this paper we introduce a new dynamic data flow analysis tuned to automate design understanding. We present the use of the approach for feature localization and for understanding the design's data flow. In the evaluation, our analysis improves feature localization by reducing the uncertainty by 41% to 98% compared to a previous approach using coverage metrics.

I. INTRODUCTION

Modern chip designs are getting more and more complex. To keep up with the complexity, the size of the design teams increases as well. Today, design teams for state-of-the-art chips already consist of hundreds of people [1]. To fulfill the time-to-market constraints, more and more third-party blocks are used or old design blocks are reused. If improvements or bugfixes are necessary, these often relate to some specific features of the design. Following the definition of the IEEE Standard 829 [2], a **feature** is a distinguishing characteristic of the design. Generally, a feature is related to some explicit or implicit parts of the design's specification. A feature is mostly defined in terms of functionality, performance, or robustness. In this paper we are primarily interested in functional features.

A designer, who wants to improve a feature or fix a bug, must commence two initial steps in order to fulfill this task. First, he must localize the place, where the feature is implemented and second, he must understand how the implementation of the feature works. Having a good documentation of all parts of the design, including third-party blocks and blocks from previous designs, clearly helps with this task. Unfortunately, in many cases there exists only a poor documentation. Consequently, the designer is required to manually inspect the code of the design until he finds code belonging to a feature. Then he still has to understand the feature's implementation.

In this paper we present a dynamic analysis of the information flow on *Hardware Description Language* (HDL)-level gathered during simulation. The only requirement for the analysis is that the HDL code of the design is available. This analysis can be used for different tasks while debugging or improving a design. First, for a single use case the analysis

serves to visualize the data-flow and control-flow in the full design or in parts of the design. This gives a designer valuable insight into the implementation of features. Second, our analysis can significantly improve the results of automated feature localization, pinpointing the parts of the design which implement a feature. Additionally, both approaches can be combined allowing a designer to localize parts in the data- and control-flow which are related to a feature.

The remainder of this paper is organized as follows: In Section II related work is presented, followed by definitions in Section III used throughout this paper. In Section IV we introduce relevant subgraphs, including a brief description how the subgraphs are computed. Section V presents how relevant subgraphs are used for design understanding. This includes feature localization and visualization of data- and control-flow within the design. In Section VI we evaluate our approach. Finally, Section VII concludes the paper.

II. RELATED WORK

Over the years several methods have been proposed to support a developer in understanding hardware and software designs. One of the first techniques was static program slicing, originally introduced for software designs by Weiser [3] and adapted for HDL-descriptions by Clarke et. al. [4]. Static program slicing was originally proposed for debugging, but can also be used for design understanding, as it removes unrelated code. Static program slicing computes all statements, which are affecting or are affected by a given set of program positions, called *slicing criterion*. The resulting slice is a subset of the original system. With respect to the slicing criterion, the slice is functionally equivalent to the original system, for all possible inputs of the system. However, as static program slices must be equivalent under all inputs, they are rather large.

To reduce the size of the slice, dynamic program slicing has been proposed for software systems in [5]. A slicing criterion for dynamic program slicing is a defined point in the execution trace of the system under specific input. The resulting slice is only required to be equivalent under that input. There exist several types of dynamic slicing [6]: *data slicing* computes only the transitive closure of dynamic data dependency. In *full slicing* also the control dependencies are considered. In *relevant slicing* additionally those statements are added to the resulting slice which would influence the value of a variable by changing the execution path. The technique presented in this paper considers dependency in the sense of full slicing.

Similar to backward dynamic program slicing, path tracing tries to determine the cause of a value in a concrete run of the

design. Path tracing of hardware designs was first introduced in [7]. In the original version, path tracing works on the gate level computing the critical path of a design by following the controlling inputs of a gate. However, in this paper we are interested in design understanding on HDL-level. In [8] an approach is described to apply path tracing on HDL-level, but that approach only considers data dependency and not control dependency. Therefore the result is, in the sense of dynamic program slicing, a backward data slice. Our approach however, can be used for forward and backward full slices.

Another approach for design understanding is feature localization using coverage metrics, first introduced for software designs in [9]. Feature localization aims to automatically find the part of the code implementing a feature. For feature localization conventionally the coverage of runs using a defined feature is compared with the coverage of runs not using that feature. There exist several heuristics for this comparison [10].

Feature localization for HDL-designs has first been applied in [10]. Toggle- and statement-coverage are used as coverage metrics for the comparison. The results are presented using a coloring scheme of the source code, inspired by the Tarantula tool [11]. The authors compare several different coloring heuristics and report that the coloring heuristic based on the Tarantula scheme yields the best results. In contrast, the technique presented here uses dynamic dependency information, rather than coverage information, allowing a more accurate localization of features.

III. PRELIMINARIES

The **dependency graph** of a design contains a vertex for each statement in the source code of the design. Two vertices are connected by a directed edge, iff there exists an input to the system such that the starting point of the edge can affect the ending point of the edge. A dependency graph of a design can be statically computed from the source code of the system.

A simulation or emulation of a hardware design under specific input, as well as the execution of a software system under specific input, is called a **run**. Throughout this paper we will use the following code as a running example:

```
1 input wire a,b,clock;        8 always @(posedge clock)
2 output reg x,y;              9   if(~a)
3 always @(posedge clock)     10     y<=x;
4   if(a)                     11   else
5     x<=~b;                  12     y<=0;
6   else
7     x<=b;
```

We consider an exemplary run with two clock cycles: in the first cycle a=1 and b=1, and in the second cycle a=0 and b=1.

During a run r of an HDL design H, a statement or expression in the source code can be executed several times. The executions can be sequential or parallel. Let an **execution point** x be one single execution of a statement or an expression, defined by its corresponding source code position, instantiation path, and time of execution. If x corresponds to an assignment statement, we call it an **assignment execution point**. Accordingly, for a control statement we call it a **control execution point** and for an expression **expression execution point**. Let X be the set of all execution points in r.

Further, let a **data point** d be the result of the evaluation of an expression, the value of a constant, or the value of an instantiated variable after an assignment to this variable[1]. Let D be the set of all data points in r.

[1]The change of a primary input and setting the initial value of a variable is also considered as an assignment in this sense.
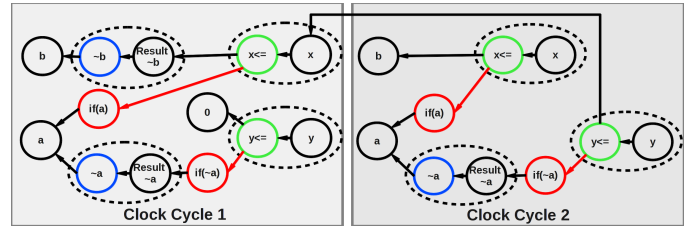

Figure 1. The dynamic dependency graph for our running example.

Given the design H and a run r, a directed and acyclic **dynamic dependency graph** $G = (V, E)$ can be computed. Figure 1 shows the dynamic dependency graph for our running example. The vertex set $V = D \cup X$ of G is the union of the execution points X and the data points D. In contrast to standard dynamic dependency graphs from literature [12], we also have data points in our vertex set because in HDL we do not have expressions reading input data. Instead the input is given by external assignments to the primary input pins. Hence, without the data points we would not have the input values in our dynamic dependency graph. In Figure 1 black circles represent data points, blue circles expression execution points, green circles assignment execution points, and red circles control execution points.

An edge $e = (v_1, v_2)$ of G is the directed connection of its starting point $v_1 \in V$ and its ending point $v_2 \in V$. The edge set E of G is defined by the direct-dependent relations of the vertices. There exist two direct-dependent relations, direct-data-dependent and direct-control-dependent. A data point $d \in D$ is **direct-data-dependent** on an execution point $x \in X$, if x is the assignment execution point which originally set d or x is the expression execution point for which d is the result. Further, x is **direct-data-dependent** on d, if d is an operand of x. An execution point $x_1 \in X$ is **direct-control-dependent** on a control execution point $x_2 \in X$, if x_2 is the control execution point which controls the execution of the basic block containing x_1. Figure 1 shows direct-control-dependent edges as red arrows and direct-data-dependent edges as black arrows.

A vertex $v_1 \in V$ is **direct-dependent** on a vertex $v_2 \in V$, if v_1 is direct-control-dependent on v_2 or v_1 is direct-data-dependent on v_2. Further, v_1 is **data-dependent** on v_2 if there exists a path in G from v_1 to v_2 only consisting of direct-data-dependent edges. A vertex v_1 is **control-dependent** on v_2 if there exists a path in G from v_1 to v_2 consisting only of direct-control-dependent edges. A vertex v_1 is **dependent** on v_2 if there exists a path in G from v_1 to v_2.

For a vertex $v \in V$ the **backwardtrace**$(v) \subseteq V$ is the set of all vertices which are reachable in G from v following the edge direction. Correspondingly, for a vertex $v \in V$ the **forwardtrace**$(v) \subseteq V$ is the set of all vertices which are reachable in G from v following the edges in reverse direction.

For each assignment execution point and each expression execution point x, there exists exactly one data point d which is direct-dependent on x. Also for every data point d, if there exists an expression point x on which d is direct-dependent then x is unique and x is either an expression execution point or an assignment execution point. We can reduce G by merging the expression and assignment execution points with their corresponding data points without losing information. In Figure 1 the vertices which can be merged are shown by dotted circles. The corresponding reduced dynamic dependency graph is shown in Figure 2. For the rest of the paper we will consider reduced dynamic dependency graphs.
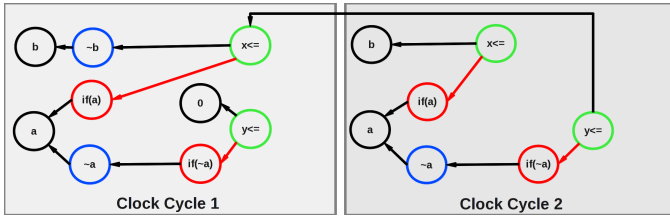
Figure 2. The reduced dynamic dependency graph for our running example.

IV. FROM DYNAMIC DEPENDENCY GRAPHS TO RELEVANT SUBGRAPHS

In a run of an HDL-design large parts of the code are always executed, but in fact only parts of that code affect the result of the run. Similarly, there exists code, which influences the result, but its effect is independent of the values that define which features are executed. For example consider a simple memory controller with no error checking which reads from a memory location. Thus, the contents of the memory affects the result of the operation, however, for the feature "reading memory content", the content is irrelevant.

In this section we describe how to compute a subgraph of a run, which includes those vertices and edges that are relevant for the result of the run. In Section IV-A we present the formal definition of such relevant subgraphs. Section IV-B describes how the subgraphs are computed.

A. Formal definition

Given a dynamic dependency graph $G = (V, E)$ we can compute a **relevant subgraph** $G_n = (V_n, E_n)$, containing those parts of the code that affect the result and those parts of the code that are affected by the input. For this, we first need a set of starting vertices $V_s \subseteq V$ and a set of ending vertices $V_e \subseteq V$. The set V_s corresponds to the values at a given point in time, deciding the features used. The set V_e represents the result of the features. In most cases the features used are chosen by the primary inputs and the result is given at the primary outputs. Therefore, V_s normally is a subset of the data points at the primary inputs and V_e is a subset of the data points at the primary outputs. Given G, V_e, and V_s we can compute the relevant subgraph G_n. A vertex $v \in V$ is contained in V_n iff v is element of a path from an element of V_e to an element in V_s, or formally:

$$\forall v \in V, [v \in V_n \Leftrightarrow$$
$$(\exists v_e \in V_e, (v \in backwardtrace(v_e)) \vee (v \equiv v_e)) \wedge$$
$$(\exists v_s \in V_s, (v \in forwardtrace(v_s)) \vee (v \equiv v_s))]$$

An edge $e \in E$ is part of E_n, iff its starting and its ending point are part of V_n, or formally:

$$\forall e \in E, [e \in E_n \Leftrightarrow (v_1 \in V_n) \wedge (v_2 \in V_n) \wedge (e \equiv (v_1, v_2))]$$

Figure 3 shows the relevant subgraph for our running example when we use the data points corresponding to the primary inputs in the first clock cycle as V_s and the data points corresponding to the primary outputs in the second clock cycle as V_e. By construction, the set of all statements corresponding to execution points in V_n is a subset of the statements covered by r. Analogously, the expressions corresponding to expression execution points in V_n are a subset of the expressions covered by r.

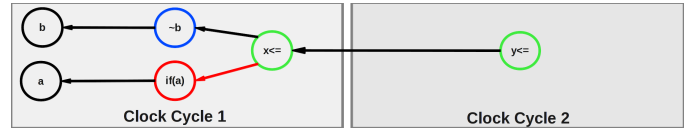Figure 3. A relevant subgraph for our running example.

B. Computation

For computing G_n we simulate an instrumented Verilog design. Initially, the elements in V_s are marked. Consecutively, the result of an expression is marked, if at least one operand is marked and changing the values of the marked operands would change the result. Additionally, we store the minimal set of marked operands, which have to change their values in order to change the result of the expression. If several minimal sets exist, we store the union of those sets.

A control execution point c is marked if its operand is marked or if the control execution point is marked, which controls the execution of the basic block containing c; the cause for marking c is also stored. The assigned value of an assignment execution point a is marked, if the operand of a is marked or if the control execution point which controls the execution of the basic block containing a is marked; the cause for marking a is stored.

Let $M \subset V$ be the set of all marked elements. After simulation, for any marked element $v \in M$ it is true that $\exists v_s \in V_s, (v \in forwardtrace(v_s)) \vee (v \equiv v_s)$. Thus, for our computation we never compute the complete dynamic dependency graph, rather we only compute the forwardtrace of the starting set. Given V_e and the marked elements M, we can compute V_n using the store information about the marking cause to determine all elements $m \in M$ for which $\exists v_e \in V_e, (m \in backwardtrace(v_e)) \vee (m \equiv v_e)$ holds.

V. RELEVANT SUBGRAPHS FOR DESIGN UNDERSTANDING

In this section we will show how relevant subgraphs can be used for design understanding. In Section V-A we show feature localization using relevant subgraphs. Section V-B presents a visualization for the relevant subgraph to help understanding the data-flow of a design.

A. Feature localization

Automatic feature localization computes the **likelihood** of code to be related to a feature. Feature localization requires runs for which it is known what features of the system are used. Conventionally, coverage metrics are used for this computation [9]. The Tarantula-based heuristic [11] computes the likelihood by comparing the percentage of runs using a feature and covering a statement with the percentage of runs not using the feature and covering the statement. This heuristic has shown good results [10]. Therefore, we adapt this heuristic for our technique.

In this paper we propose relevant subgraphs as basis on which the likelihood is computed, instead of using coverage metrics. For this, we first compute the relevant subgraph for each run. Second, for each subgraph we compute the set of expressions and statements corresponding to execution points in the subgraph. Finally, we use those sets as input for the heuristic to compute the likelihood of statements and expressions to be related to a feature. The likelihood is presented to the user by color-coding the source code.

Several likelihoods may be associated to an expression or statement, because a statement can have one or more expressions and an expression may consist of sub-expressions.
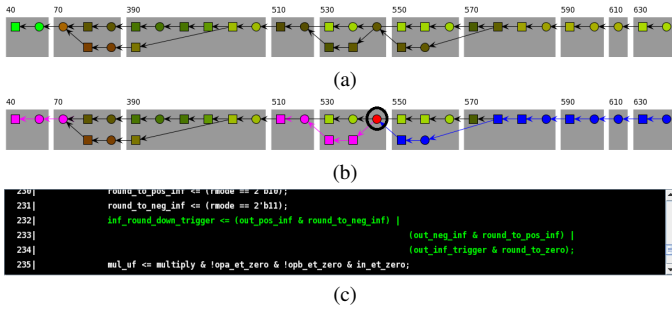
Figure 4. (a) A relevant subgraph, visualized by our tool. (b) Selecting a vertex highlights its backwardtrace and forwardtrace. (c) Shows the corresponding source code of the selected vertex.

The resulting color of a character in the source code is defined by the likelihood of the smallest statement or expression, containing the character.

This enables the user to distinguish those cases where a statement or expression is necessary for the feature, but not all of its sub-expressions. To illustrate this, consider the following code snippet taken from one of our case studies:

```
inf_round_down_trigger <=
    (out_pos_inf & round_to_neg_inf) |
    (out_neg_inf & round_to_pos_inf) |
    (out_inf_trigger & round_to_zero);
```

This code from a floating-point-unit decides if an infinite value should be rounded to a concrete number. Based on the rounding-mode requested, the condition applied for this decision is different. Each part of the disjunction corresponds to the condition for a different rounding mode. Such that the assignment is part of each of those rounding modes. However, for each rounding mode there is exactly one sub-expression which is related to the rounding mode.

In [10] automatic file ranking is described, which has been adapted in our approach. The file ranking gives the user an overview of the files belonging to a feature. The file ranking is computed by ranking those files highest, which contain source code with the highest likelihood to belong to a feature. If there are files with source code, with equal high likelihoods, the file is ranked highest which relatively contains more code with such a likelihood.

B. Presenting the graph

A further use for relevant subgraphs is visualizing the data-flow within the design. Such a visualization can further help in understanding the design. Especially, we combine the presentation with feature localization, such that the different vertices in the graph are colored corresponding to their likelihood to be related to a feature. Hence, our presentation of the graph supports the user to find features within the design's data-flow.

Figure 4 illustrates an example of a relevant subgraph. Figure 4(a) shows the graph. The vertices are colored with respect to their likelihood to be part of the feature. Vertices with the same time of execution are grouped together by gray rectangles. The time is shown above the rectangles. As a clock cycle, depending on the use case, can take more than one time unit, or a value may stay inside a register for several clock cycles before it is read again, only that points in time are shown which contain vertices. To simplify navigation through the graph, the backwardtrace and forwardtrace of a selected vertex are highlighted (Figure 4(b)). As the graph by itself does not help a designer to understand the HDL code, the source code corresponding to the selected vertex is shown to the user (Figure 4(c)).

Table I
DESIGNS USED IN THE CASE STUDY.

Design	LOC	Files	use cases	features
double_fpu_verilog	2555	7	320	8
SD/MMC Controller	3840	17	5	5

Table II
FILE RANKING FOR DOUBLE_FPU_VERILOG COMPARED TO DOCUMENTATION.

Feature	documentation	coverage	tracing all inputs	tracing control inputs	tracing single input
Addition	fpu_add fpu_sub	fpu_sub fpu_add	fpu_double fpu_add fpu_sub	fpu_double fpu_sub fpu_add	fpu_double fpu_sub fpu_add
Substraction	fpu_sub fpu_add	fpu_double fpu_sub fpu_add	fpu_double fpu_add fpu_sub	fpu_double fpu_sub fpu_add	fpu_double fpu_sub fpu_add
Multiplication	fpu_mul	fpu_mul	fpu_mul	fpu_mul	fpu_mul
Division	fpu_div	fpu_div	fpu_div	fpu_div	fpu_div
round to nearest even	fpu_round fpu_exceptions	-	fpu_exceptions fpu_round	fpu_exceptions fpu_round	fpu_exceptions fpu_round
round to zero	fpu_round fpu_exceptions	-	fpu_exceptions fpu_round	fpu_round fpu_exceptions	fpu_round fpu_exceptions
round to +INF	fpu_round fpu_exceptions	-	fpu_exceptions fpu_round	fpu_exceptions fpu_round	fpu_exceptions fpu_round
round to -INF	fpu_round fpu_exceptions	-	fpu_exceptions fpu_round	fpu_exceptions fpu_round	fpu_round fpu_exceptions

VI. EVALUATION

For our evaluation we used two designs from the OpenCores.org-website. Table I gives an overview over the designs. Due to space limitation we compare in detail feature localization using our analysis to feature localization using coverage metrics [10], but we omit the evaluation for the graph presentation.

A. Design: double_fpu_verilog

The floating-point-unit double_fpu_verilog provides eight features in total: four arithmetic operations and four rounding modes. An arithmetic operation and a rounding mode must always be used together, hence, there are totally 16 different combinations of features. An operation takes up to 71 clock cycles and the design indicates the end of the operation by asserting a ready signal. For the use cases we used 20 different sets of operand values, 14 randomly chosen and 6 chosen such that they contain special values (zero, denormalized numbers, infinite, near infinite). Each of the input sets is applied to each of the 16 different combinations of features, resulting in a total amount of 320 use cases.

We also evaluated how the marking of different inputs affects the quality of the result. For this, we conducted the case study with different starting sets. For the double_fpu_verilog design, there are special primary input pins to choose the arithmetic operations and the rounding mode. The first starting set contains all data points corresponding to primary inputs. The second set contains all data points corresponding to primary inputs, which define the features to be used. The final set contains all data points corresponding to the single primary input, which defines the usage of a certain feature. For the last set we executed each use case twice. Once, with only data points in the starting set corresponding to primary inputs, which decide the arithmetic operation, and once, with only data points corresponding to primary inputs deciding the rounding mode. For all cases, as ending set we used the values at the primary outputs after the design has indicated the end of the operation.

Improving file ranking

First, we compared file ranking of feature localization using our analysis to coverage metric based file ranking. The result of this comparison is shown in Table II. We only consider files for file ranking with likelihoods larger than 0.5. Otherwise for

Table III
REDUCTION OF UNCERTAINTY FOR DOUBLE_FPU_VERILOG.

Feature	Starting set containing all inputs										Starting set containing feature inputs										Starting set containing single input									
	common by coverage					uncertain by coverage					common by coverage					uncertain by coverage					common by coverage					uncertain by coverage				
		related		unrelated			related		unrelated			related		unrelated			related		unrelated			related		unrelated			related		unrelated	
	#stm	#	%	#	%	#stm	#	%	#	%	#stm	#	%	#	%	#stm	#	%	#	%	#stm	#	%	#	%	#stm	#	%	#	%
Addition	330	0	0%	270	82%	413	3	<1%	310	75%	330	0	0%	317	96%	413	1	<1%	368	89%	330	0	0%	324	98%	413	1	<1%	375	91%
Subtraction	314	0	0%	262	83%	397	0	0%	305	77%	314	0	0%	304	97%	397	1	<1%	358	90%	314	0	0%	311	99%	397	1	<1%	365	92%
Multiplication	368	0	0%	308	84%	390	1	<1%	317	81%	368	0	0%	342	93%	390	1	<1%	358	92%	368	0	0%	349	95%	390	1	<1%	367	94%
Division	285	11	4%	231	81%	336	13	4%	259	77%	285	0	0%	278	98%	336	2	1%	313	93%	285	0	0%	285	100%	336	2	1%	320	95%
Round to -infinite	626	1	<1%	263	42%	637	1	<1%	264	42%	626	1	<1%	391	62%	637	1	<1%	392	62%	626	0	0%	620	99%	637	0	0%	622	98%
Round to +infinite	619	0	0%	257	42%	638	0	0%	264	41%	619	0	0%	384	62%	638	0	0%	392	61%	619	0	0%	613	99%	638	0	0%	621	97%
Round to zero	620	0	0%	262	42%	633	0	0%	268	42%	620	0	0%	389	63%	633	0	0%	396	63%	620	0	0%	615	99%	633	0	0%	621	98%
Round to nearest even	634	1	<1%	265	42%	638	1	<1%	265	42%	634	0	0%	389	61%	638	0	0%	391	61%	634	0	0%	618	97%	638	0	0%	620	97%

the coverage metric based approach all files would be included and the ranking of the additionally added files would only depend on how much of the code is always executed, such that the ranking would be independent of the feature considered.

For the arithmetic operations, both approaches determine the modules the documentation relates to the feature. *Addition* and *subtraction* are special cases in the design. Based on the signs of the operands, a *subtraction* can be executed in the *addition* unit and vice versa. The decision, which unit is used to execute the operation, is made in the top module (`fpu_double`). Hence, for all subgraph-based experiments the `fpu_double` is ranked highest because the corresponding code is always in the relevant subgraph when the feature is used. For *addition* the coverage metric based approach does not relate this code to the feature, considering these parts of the code commonly used for all features.

When comparing the different starting sets, the main difference is that feature localization, using the starting set with all inputs, relates some reset code for internal registers to the arithmetic features. However, this code is considered unrelated to any feature using the other starting sets.

When considering the rounding mode features, the coverage metric based approach does not rank any files. In contrast, for all starting sets the file ranking of the subgraph-based approach yields exactly those files, the documentation relates to the features. Even as the different starting sets yield the same file ranking, there are differences between them, when considering the result on statement and expression level. For the rounding modes, the design first checks if any type of rounding should be performed. Then the concrete rounding is applied using identical code for all rounding modes. When comparing the starting set, including all inputs, with the set, only including inputs deciding the features, the first one marks parts of the code as commonly used for all features, which the second set does not consider at all. The third set improves the result, as it distinguishes between the code common for all rounding modes and code not related to rounding at all.

Detection of irrelevant code

Next we measured how many of the statements, which the coverage metric based approach relates to features, our approach has shown to be unrelated to any features. There are 641 statements which the coverage metric based approach relates to some feature. Using the starting set with all inputs, 263 of those statements are determined to be unrelated to any feature. This is a reduction by 41%. The statements removed in this way can be categorized into two groups. The first group contains statements, resetting registers. However, those resets only affect the behavior of the design at a point where the specification has no restriction upon the affected outputs. The second group belongs to the handling of *Quiet Not a Number*-values (QNaN) as operand values. In such a case the design

immediately outputs QNaN without executing any arithmetic or rounding operation. Hence, this group is clearly not part of the desired features.

The other two starting sets remove 392 or 61% of the statements, respectively. The additionally removed statements are remaining parts of the reset- and initialization-code. This shows nicely that the starting set should be chosen based on the feature definition: If reset and initialization are defined as a part of the feature, the reset input should be included to the starting set. However, if the feature is defined as a pure computation, the reset signal should be excluded from the starting set.

Reduction of uncertainty

In addition, we measured how much of the code, which the coverage metric based approach considers as part of all features, our approach can decide if it belongs to a feature or not. We perform this comparison on statement level. First, we need comparable likelihoods between the two types of feature localization. To relate both approaches to statement level, we used the following rules:

1) For our approach we used the maximum likelihood of a statement and its sub-expressions.
2) For the coverage metric based approach statements with a likelihood lower than 0.5 by statement coverage, the likelihood by statement coverage is used. Otherwise, the maximum of the likelihood by statement coverage and all likelihoods by toggle coverage for variables used in the statement is applied.

The intention behind the first rule is that a statement commonly used for all features, but with a sub-expression specifically used by one feature, is worth inspecting. And a statement used by a feature with one sub-expression not used by the feature, is still part of the feature[2]. With the second rule we followed the observation in [10] that statement coverage gives a broad overview and toggle coverage allows to distinguish code, which statement coverage considers as commonly used for all features. Thus, our implementation for the coverage based feature localization is an improvement in comparison to the approach presented in [10] as it removes the inconsistency between the two different coverage metrics.

Table III shows the results of measuring the amount of code our approach can relate or exclude from a feature, which the coverage metric based approach could not classify. Column 1 gives the name of the feature. Columns 2-11 give the comparison with the starting set containing all inputs. Here, Column 2 gives the number of statements which the coverage metric based approach considers to be shared with the other features (likelihood equals 0.5). In Column 3 we give the absolute number of statements for which our approach is

[2]Note the case that a statement is not used for a feature but one of its sub-expressions is impossible by the construction of subgraphs.

highly confident, that they belong to the feature (likelihood larger than 0.9), but the coverage metric based approach relates to all features. Column 4 gives the percentage of those statements. Column 5 gives the absolute number of statements, which our approach is highly confident that they are unrelated to a feature (likelihood less than 0.1 or irrelevant to all features), and the coverage based approach relates to all features. Column 6 gives the relative percentage of those statements. Additionally, in Columns 7-11 we consider code where the coverage metric based approach is non-conclusive if the code is related to a feature or not (likelihood between 0.25 and 0.75). Again, we checked how many of those statements our approach can definitely relate to a feature (Columns 8-9) and how much of those code our approach shows to be unrelated to the feature (Columns 10-11). Columns 12-21 show the results of the starting set containing only the input values selecting the desired features and Columns 22-31 show the results for the starting set containing only one single input value.

The feature localization using our approach reveals for a large amount of code, considered as common to all features by the coverage metric based approach, that it is unrelated to a feature. Since completely unrelated code needs not to be considered, feature localization using relevant subgraphs reduces the code which a designer has to inspect, when improving or bugfixing a feature.

B. Design: SD/MMC Controller

For the second evaluation we used the design `SD/MMC Controller`. This design is a controller for an SD-card connected to the controller via an SPI-bus. Besides the primary inputs and outputs for the SPI-bus, the controller implements a Wishbone-interface for receiving commands and sending data. As use cases we utilize the original test bench of the design.

Again, we compare coverage metric based feature localization with feature localization using relevant subgraphs and different starting sets. The first starting set contains all primary inputs excluding the clocks. The second starting set contains only the inputs of the Wishbone-interface, such that the contents on the SD-card are not part of the starting set. In all cases, as ending set the values of the primary outputs are used. However, the first and second starting set result in exactly the same likelihood computed for every source code position. Out of this reason, in the following only the results for the first starting set are discussed.

For three of the features provided by the design, the file ranking of the two approaches are identical. For the other two features, the documentation of the design was not clear enough to decide which of the two approaches yields a better result.

In this evaluation again much of the code which feature localization using coverage metrics relates to some feature, is shown to be unrelated to any feature by our approach. In numbers these are 354 out of 767 statements, or a reduction by 46%. The reason is that the design assigns a value to all registers during a clock cycle. In many cases these statements are placed in deeply nested conditions related to a feature. For the coverage metric based approach those statements are considered as part of the feature. However, in most cases the value is never read before the register gets a new value assigned. In contrast, our approach is able to detect this.

As for the previous design, we measured how much of the code, which the coverage metric based approach considers as part of all features, can be related to a feature or excluded

Table IV
REDUCTION OF UNCERTAINTY FOR SD/MMC CONTROLLER.

| Feature | common by coverage | | | | | uncertain by coverage | | | | |
| | | related | | unrelated | | | related | | unrelated | |
	#stm	#	%	#	%	#stm	#	%	#	%
Register access	89	1	1%	84	94%	115	1	1%	108	94%
SPI Bus access	74	2	3%	59	80%	191	3	2%	118	62%
Init SD Card	56	0	0%	55	98%	235	0	0%	229	97%
SD card read	40	0	0%	39	98%	217	3	1%	163	75%
SD Card write	40	9	23%	30	75%	231	19	8%	108	47%

from that feature, respectively. The result of this measurement is given in Table IV.

On average 94% of the code which feature localization using coverage metrics considers as part of all features can definitely be related to a feature or excluded from the feature. For code where feature localization using coverage metrics is non-conclusive, our approach still can decide for on average 78% if the code belongs to the feature or not. Overall, our approach can exclude a large amount of code from consideration, which feature localization with coverage metrics includes.

VII. CONCLUSION

In this paper we present a new data flow analysis approach for HDL-designs based on dynamic dependency graphs. We showed two application scenarios for design understanding: feature localization and graph extraction for dynamic data flow within the design. The second application can be used to navigate through the source code along the designs data-flow. Additionally, both scenarios can be combined allowing the user to find parts of the data-flow relevant to a feature.

We compared feature localization using our analysis against feature localization using statement- and toggle-coverage. The evaluation clearly shows the advantage of our approach, reducing uncertainty by 41% up to 98% for the considered designs.

The presentation of the data flow graph improves design understanding as it nicely shows the relation between different statements, not clear from the pure code alone.

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] M. Weiser, "Program slicing," in *Proceedings of International Conference on Software Engineering*, 1981, pp. 439–449.
[4] 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.
[5] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, pp. 155 – 163, 1988.
[6] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," in *Proceedings of International Symposium on Automated analysis-driven debugging*, 2005, pp. 33–42.
[7] M. Abramovici, P. R. Menon, and D. T. Miller, "Critical path tracing - an alternative to fault simulation," in *Proceedings of Design Automation Conference*, 1983, pp. 214–220.
[8] M.-C. Lai, C.-H. Lee, B.-H. Ho, and J.-S. Tsai, "Active trace debugging for hardware description languages," US Patent 6 546 526, 2003.
[9] 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.
[10] 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.
[11] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for fault localization," in *Proceedings of the Workshop on Software Visualization*, 2001, pp. 71 –75.
[12] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *ACM SIGPLAN Notices*, vol. 25, no. 6, pp. 246–256, Jun. 1990.