Chapter 2
Evaluating Debugging Algorithms from a Qualitative Perspective[*]

Alexander Finder and Görschwin Fey

Abstract A bottleneck during hardware design is the localization and the correction of faults – so-called *debugging*. Several approaches for automation of debugging have been proposed. This paper describes a methodology for evaluation and comparison of automated debugging algorithms. A fault model for faults occurring in SystemC descriptions at design time or during implementation is an essential part of this methodology. Each type of fault is characterized by mutations on the program dependence graph. The presented methodology is applied to evaluate the capability of a simulation based debugging procedure. Both qualitative and quantitative assessments are made to evaluate the fault model.

2.1 Introduction

During design of *Very Large Scale Integrated* (VLSI) circuits often functional mismatches between a given specification and the final implementation occur. When an implemented design produces erroneous output due to the presence of one or more faults, debugging begins. First sophisticated automatic approaches for debugging exist [4, 9, 11, 13] and several further diagnosis techniques have been developed, e.g. [6, 14].

So far the work on comparing these approaches and on understanding which types of design bugs can be efficiently handled by a certain approach is very

A. Finder (✉) • G. Fey
Group of Computer Architecture, University of Bremen, Bremen, Germany
e-mail: final@informatik.uni-bremen.de; fey@informatik.uni-bremen.de

limited. For instance, in [3, 10] different debugging approaches were compared. In [10] a procedure based on explanation is compared to a model-based diagnosis technique. The comparison is mainly done on the basis of a case study. In [3] a simulation-based diagnosis technique and a diagnosis technique based on *Boolean Satisfiability* (SAT) are compared. There the quality of the two techniques is quantitatively assessed and compared by measuring the distance between gate level fault candidates and actual faults. No generalization to the source level, e.g. in a *Hardware Description Language* (HDL) has been done. Also the work in [11] quantitatively assesses a debugging algorithm by measuring the distance between actual fault sites and candidate fault sites determined by the algorithm. By this, all of these approaches and the conclusions drawn are restricted to the respective benchmarks considered. Generalizing the results is difficult.

One way towards generalizing the result is the use of fault models to assess the performance of an algorithm for certain types of design bugs. No appropriate fault model has been introduced so far. Previous fault models have been developed for other purposes. Fault models known from testing integrated circuits for production faults are efficient in modeling physical failures, like e.g. the stuck-at fault model [7]. They are not applicable when considering design bugs. A fault model on the netlist level has been proposed in [1] to capture faults introduced after synthesizing HDL descriptions. Additionally, high-level fault models have been introduced. For example, in [5] a fault model is described for determining bit coverage information. The fault model for SystemC presented in [2] describes transient and permanent faults. These previous fault models cannot be used for describing bugs at the HDL level.

In this paper a methodology is presented to evaluate debugging algorithms from a qualitative perspective. As a basis we use an extensible fault model that describes different types of bugs in SystemC descriptions. Some parts of the model are inspired by previous work from Abadir, Ferguson, and Kirkland [1]. We lift this fault model originally defined for gate level netlists to higher level descriptions. Based on this fault model, debugging algorithms can be assessed to understand their capabilities with respect to different types of bugs.

In a first case study we show that some types of bugs can be handled using a simulation based algorithm while other types of bugs cannot be handled. By this, our methodology qualitatively classifies the debugging algorithm. Knowing such restrictions is important from two points of view: (1) the results returned by the debugging algorithm may be misleading for those bugs that cannot be handled, (2) a comparison to other debugging algorithms becomes possible. We will also discuss why using a quantitative approach like in [3, 11] is difficult and requires further research before a generalization of the results is possible.

The contributions of the presented work are

- a methodology for evaluating debugging algorithms,
- a fault model on the HDL level to classify design bugs, and
- a discussion and evaluation of a quantitative approach to assess debugging algorithms.

This paper is structured as follows. In Sect. 2.2 a short introduction to source code analysis and simulation-based debugging is given. The general idea underlying this paper is described in more detail in Sect. 2.3. Furthermore, this section also discusses the problems of quantitative approaches when evaluating debugging algorithms. Section 2.4 explains the proposed fault model for bugs in SystemC designs offering a possibility for evaluation and comparison of debugging methods. In Sect. 2.5 the applicability and accuracy of the debugging procedure for SystemC designs is evaluated using the formerly described fault model. In Sect. 2.6 we give a conclusion.

2.2   Preliminaries

In this section some essentials of source code analysis are briefly reviewed. In particular, terminology used in this paper, *program dependency graphs* (PDGs), and simulation-based debugging are considered.

2.2.1   Faults, Bugs, and Errors

Throughout this paper we consider a *bug* to be contained in some design description. An *error* is the observation of the effect of a bug that contradicts the specification. The input stimuli leading to an error are called a *counterexample* (wrt. the specification). A *fault* is part of a fault model and, by this, a generalized description of a bug. Note, that the errors caused by a certain bug may be of various types. For example, having a wrong operator – an addition instead of a subtraction – in a computation is a typical bug. One potential error caused by this bug is an erroneous outcome of a computation. An alternative error due to the same bug in some other context may be a deadlock of concurrent processes because some resource is never released.

2.2.2   Computation of CFG and PDG

A *Control Flow Graph* (CFG) is a directed graph where the nodes represent the statements and the edges depict the control flow. The annotation at each node describes the variables defined, written or read.

Out of the CFG the *Data Dependency Graph* (DDG) can be computed. The DDG is a directed graph where the nodes indicate the statements of the program and the edges represent the dependencies between variable usages by different statements.

Further the *Control Dependency Graph* (CDG) can be computed out of the CFG. This is a directed graph where the nodes are statements and the edges depict dependencies between the statements.

Fig. 2.1 Program

```
void main (int n) {
    int i = 1;
    int sum = 0;
    while (i<n) {
        sum = sum+i;
        i++;
    }
    print (sum);
}
```
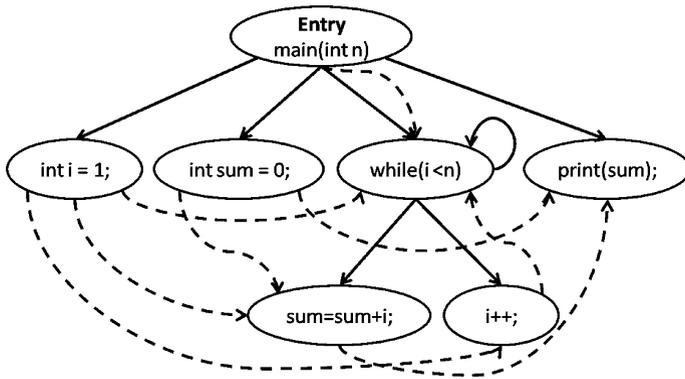

Fig. 2.2 Program dependency graph

The PDG is obtained by merging the DDG and the CDG. A PDG is a directed graph $G = (V,E)$ in which a node $v \in V$ is a statement or a predicate expression and the edges $e \in E$ incident to a node represent both, the data values the operation of the node depends on and the control condition the execution of the operation depends on. In Fig. 2.1 an example program is depicted and the corresponding PDG is shown in Fig. 2.2. Solid lines reflect control edges and dashed lines data flow.

2.2.3 Simulation-Based Debugging

Simulation-based debugging is intended to investigate the effect of statements on a variable or the influence of a variable on other statements. Simulation-based procedures are used in different areas of application, e.g. debugging, testing, compiling. In this work the simulation-based algorithm is used as a case study for the proposed methodology for assessing debugging algorithms.

The objective of the procedure is to reduce the debugging effort by focussing the attention of the user on a subset of program statements called *traces* which
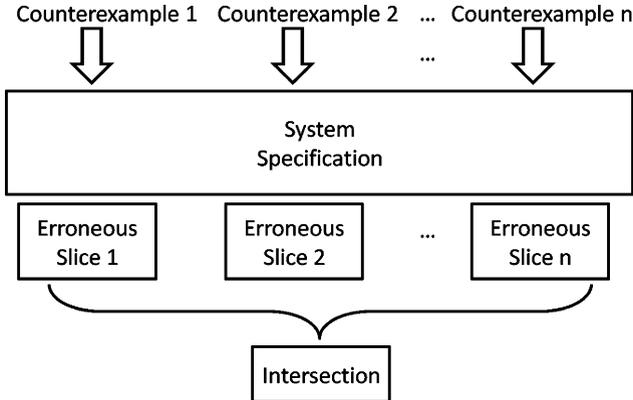
Fig. 2.3 Principle of the algorithm

are expected to contain faulty code [3]. The principle of the algorithm is shown in Fig. 2.3. For a given SystemC specification counterexamples are simulated to generate traces. The intersection of these traces includes and localizes the faulty statement. However, this is assured only if the design contains only a single bug.

2.3   General Idea and Discussion

The debugging process is comprised of collecting information from the failed simulation trace or counterexample and analyzing the design until the error source is identified. In the meanwhile several debugging algorithms and strategies exist but comparing the algorithms is difficult. Typically, even the types of bugs that can be detected by a certain algorithm are not clearly known. Thus, interpreting the output of the algorithm is hard and may even be misleading.

The idea of this paper is to use a qualitative methodology for evaluating debugging algorithms based on a fault model. This fault model induces a classification of design bugs into different types. A classification of design bugs supports to identify an existing fault type and to restrict the number of fault candidates. Using the fault model, the applicability of debugging algorithms for certain bugs can be evaluated.

Figure 2.4 outlines the relation between the faults and the design. A fault described by the fault model is a generalized description of actual bugs in a design. Each type of fault in our fault model characterizes a set of mutations of the PDG. If a faulty PDG is mutated such that the fault is rectified, the resulting PDG corresponds to a correction of the bug in the design.

Fig. 2.4 Relation between
fault and design
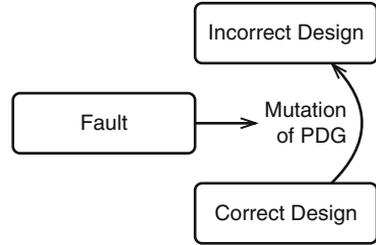
2.3.1 Qualitative Assessment

The fault model can be used to inject different types of faults in a system description. After that debugging algorithms can be assessed by the types of faults they detect and the fault candidates they return. Note that different bugs as well as corresponding programs may be functionally equivalent.

Example 2.1. Consider an operation `a+b` where `b` is faulty and the result is assigned to a variable `temp` further used in a condition. In this case we have a *data operation fault*. If the operation `a+b` is directly inserted in the condition without using `temp` we have a *control operation fault*.

This implies that a fault *A* may be transformed to a fault *B* without changing the functionality of the underlying design. Particular debugging algorithms may only be able to help in one of these cases. The use of a fault model helps to identify such restrictions of a debugging algorithm.

2.3.2 Limits of Quantitative Assessments

Extending the proposed qualitative assessment of debugging algorithms by a quantitative aspect is possible. For example, the works in [11] and [3] use distance measures between the actual fault sites and the candidate fault sites returned by the algorithms. In [11] Renieres and Reiss describe a methodology to quantitatively assess the quality of the debugging algorithm based on the PDG. In order to measure the success of a debugging algorithm the method assigns a score to the report of a fault localization, depending on the size of the report and the distance to the actual fault. Here, proximity to the fault is defined based on the PDG. In [3] the authors measure the nearest distance of a gate level fault candidate in a circuit to an actual fault, i.e. the number of gates on a shortest path to an error.

In both cases, the quantitative analysis directly depends on the benchmarks considered. The same debugging algorithm may yield very different results for the same type of bug if the benchmark changes. For example, consider one data-dominated

design performing a computation like a filter operation and a second control-dominated design containing many conditional branches. In the data-dominated design changing an operator almost always influences the output. In the control-dominated design, the output only becomes erroneous under certain conditions on the control path. The cause of the error (the bug) can be pinpointed much better in the control-dominated design. We will also show this in the evaluation of our methodology in Sect. 2.5.

2.4 Fault Model

In this section all types of faults are described and categorized that are covered by the proposed fault model. In general, faults are caused by specification changes, bugs in automated tools, and the human factor [8]. In the presented model local code transformations are considered as programming faults whereas global code transformations are considered to be design faults. As mentioned before the various types of faults do not need to be disjoint but may overlap, e.g. operator faults and predicate faults. Syntactical bugs are not classified within the proposed fault model because this kind of bugs is assumed to be discovered by a compiler, like e.g. a missing declaration or a forbidden use of a certain data type. This means that only semantic and conceptual faults are taken into account.

The proposed fault model is not claimed to be complete but maintains a list of typical faults. The applicability of this fault model has been investigated on SystemC descriptions. However, the fault model can be extended to encompass additional types of bugs, not covered so far, if needed. In the context of this work a fault corresponds to certain modifications of the PDG.

Figure 2.5 gives a hierarchical overview of the fault model described in the following sections. On the top level *programming faults* and *design faults* are distinguished. These are refined then to more concrete mutations of the PDG.
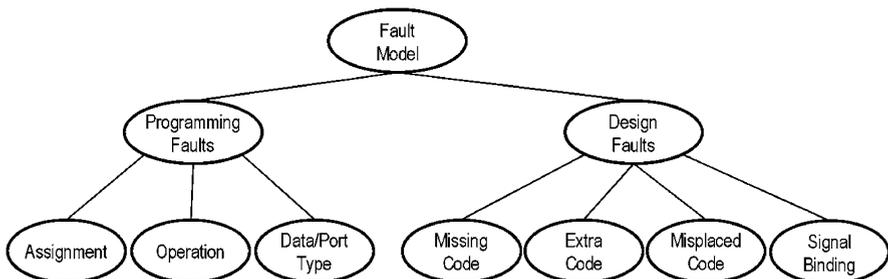

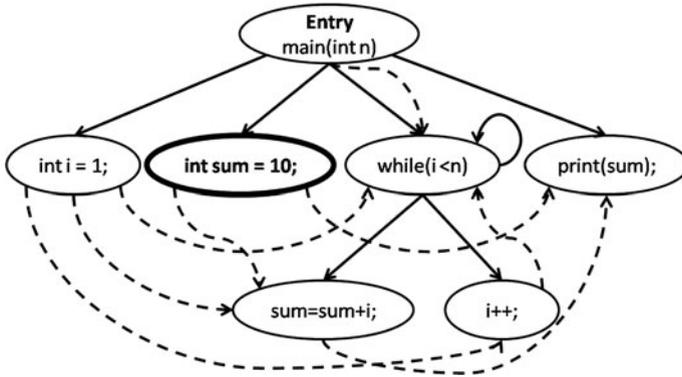
Fig. 2.5 Hierarchical view on the fault model

Fig. 2.6 Correcting an assignment

2.4.1 Programming Faults

Programming faults in SystemC specifications are assumed to be introduced during the coding phase. In the following subsections possible programming faults are described. The effect of a single fault on the PDG for the design is usually small. This is exemplarily shown for some types of faults. In all following examples we refer to the program given in Fig. 2.1.

2.4.1.1 Assignment Fault

Suppose that a wrong value is assigned to a variable. This could be done by assigning a wrong constant or a wrong variable. As a result calculations in data dependent nodes are carried out with incorrect values which may lead to erroneous output data. If a wrong constant has been assigned only one node of the PDG for the SystemC description has to be changed to fix the bug. In the other case if a wrong variable is assigned also the corresponding data edges have to be reconnected.

In Fig. 2.6 it is assumed that the programmer has inadvertently assigned the value 0 to sum but should assign 10. The effect of the correction on the PDG is indicated by bold lines.

2.4.1.2 Operation Fault

A fault is considered as an operation fault if either an incorrect data operation or an incorrect control operation is carried out. Each type of operation fault can be further partitioned in an operand fault and an operator fault (see Fig. 2.7). Depending on which type of operation fault is present, the correction of the fault has a different

Fig. 2.7 Operation fault
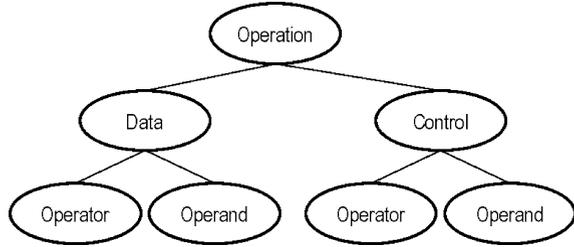
effect on the PDG. If an operator fault exists, the correction corresponds to the modification of a single node in the PDG. If the operands are wrong, also data edges have to be reconnected.

Data Operation

A data operation fault within a statement occurs if a data operator is replaced by another operator or if incorrect operand values are used within the operation. All operators defined in SystemC (+, −, *, /, %, &, —, etc.) are considered as data operators. Bugs corresponding to this fault are, e.g. using multiplication instead of division. Operands could be either variables or constants.

Control Operation

Suppose that a programmer inadvertently writes an incorrect control condition. This could be done by using incorrect operators or operands in the expression specifying the condition. There are several types of control operation faults possibly affecting the execution of a design. Writing a faulty predicate in a simple if-statement either leads to not executing the then-branch while it should be taken, or executing it, while it should not be taken. Additionally, control operation faults can be injected in loop-statements or in function calls. A faulty loop-statement leads to unspecified executions of the loop. A fault in a function call implies erroneous data.

2.4.1.3 Incorrect Data/Port Type

Suppose that the programmer declared a variable with a wrong data type. For example, the variable is of type `unsigned integer` instead of `integer` or `integer` instead of `double` and so on. This would create erroneous results in computations.

A similar fault is declaring an incorrect port type (`in`, `inout`, `out`) to a port of the system specification and binding the correct signal to the port. This would coincidently lead to missing inputs and extra outputs or vice versa to extra inputs and
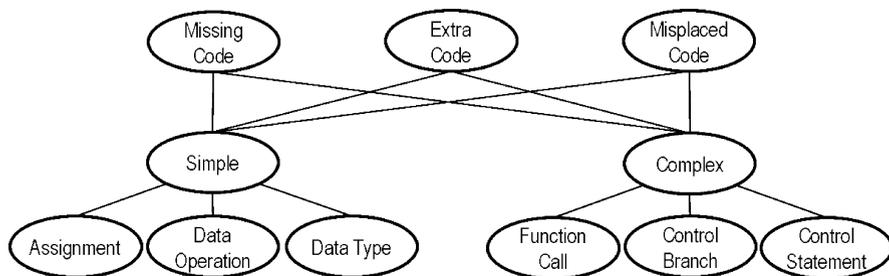
Fig. 2.8 Design faults

missing outputs. Note, a fault of this type is a SystemC specific fault and is typically not reproducible in other hardware description languages like Verilog, where the compiler detects the mismatch.

The correction of these faults would have little effect on the corresponding PDG because only the content of the nodes concerned has to be changed.

2.4.2 Design Faults

Design faults inside a given SystemC specification are expected to be introduced during the conceptual design phase. Here, we distinguish between simple and complex missing, extra, or misplaced code and signal binding faults. In Fig. 2.8 we show what may be simple and complex code parts. For instance, a single data operation or assignment is considered as simple code while function calls and control operations affecting more control and data dependencies within a PDG are treated as complex code. Here, missing code, extra code, and misplaced code can be further partitioned into the same simple and complex fault types. In the following subsections possible design faults of SystemC designs are described and the effect on the corresponding PDG is explained.

2.4.2.1 Missing Code

Similar to a missing gate or a missing inverter in gate level design [1], there could be omitted code in SystemC descriptions. Here missing simple code and missing complex code are distinguished.

Missing Simple Code

Suppose that the designer has inadvertently omitted an operation corresponding to a simple missing data operation in the SystemC implementation. The correction of
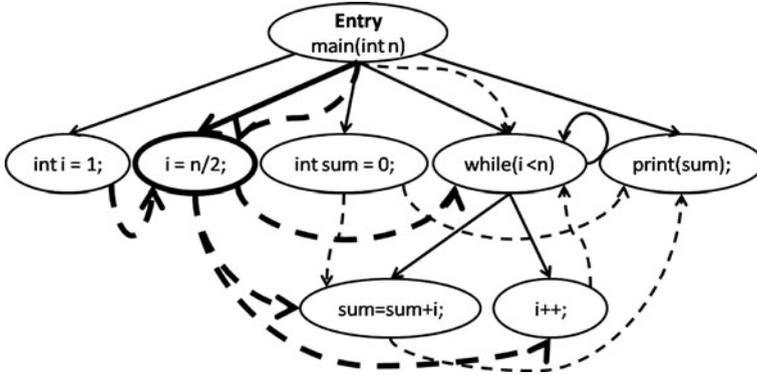
Fig. 2.9 Inserting additional code

this fault is more sophisticated than the correction of programming faults because it implies adding a node to the PDG and accompanying control and data edges to or from other nodes. Also already existing edges may have to be reconnected.

In Fig. 2.9 it is assumed that the designer omitted the statement i=n/2. The insertion of this statement implies adding a new node and a new control edge as well as adding and removing several data edges. All parts concerned are marked in bold in the figure.

Missing Complex Code

Similar to the previous design fault, a designer could omit more complex code. For instance, this may be a function call, an else-branch, or a missing control statement in terms of an if-condition or a loop-condition, embracing a block of statements. The correction of such a fault would have a large effect on the PDG. Conceivably many nodes and edges have to be added to the existing PDG restructuring the graph. Furthermore, many existing nodes and edges have to be reconnected.

2.4.2.2 Extra Code

Assume, that the designer has inserted extra simple or complex code complementary to the missing code described in the previous section. This would lead to superfluous computations or wrong control and data paths distorting the results.

The correction involves removing the extra code from the specification resulting in removing nodes and edges from the PDG. Assume, that the designer inadvertently added the extra statement sum=sum+i to the initial PDG in Fig. 2.2. In Fig. 2.10 the PDG is shown after removing the extra statement.
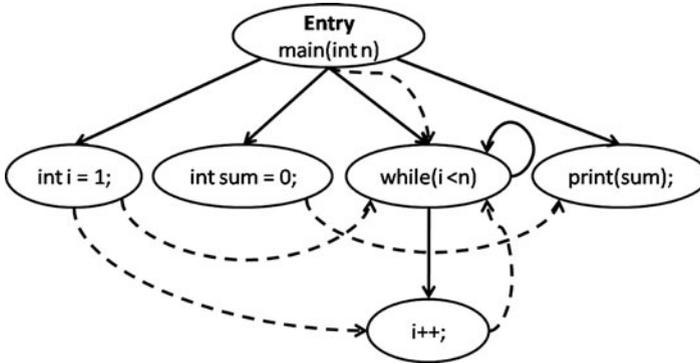
Fig. 2.10 Removing extra code

2.4.2.3 Misplaced Code

Similarly to missing or extra code, suppose that code within a specification is misplaced. This means that some statements, function calls, loops etc. will be executed before others so that we may have a faulty data or control flow within the PDG. A correction would imply reconnecting nodes in the PDG because of correcting the data or control flow.

2.4.2.4 Signal Binding Faults

Data transfer between modules is reflected by signal bindings. Each port should be bound to a certain signal. Signal binding faults may occur during the design phase. In the PDG, a correction often can be done by reconnecting data edges.

Incorrect/Interchanged Signal Binding

Suppose that the designer has specified a wrong data transfer behavior between modules leading to a wrong signal binding at a port or interchanged signals between ports. This implies incorrect data at inputs or outputs of a certain module in the system design. In the PDG we typically would have an incorrect data flow.

Missing Signal Binding

A missing signal binding means that on some arbitrary module a signal binding to a port has been omitted. That means there exists an input reading no data or an output writing no data although the data of the ports is needed in further computation steps. Note, this fault is comparable to missing simple code because the statements of the signal bindings have been omitted.

2.5   Evaluation: Simulation-Based Debugging

To evaluate our methodology, the simulation-based debugging algorithm described in Sect. 2.2 has been implemented. In our experiments, first the limitations of quantitative analysis are evaluated. Then we show the results of the qualitative assessment.

Faults in a design are localized by computing several traces which cause the program to produce erroneous output during simulation. Each trace describes a certain assignment of input variables.

Out of the SystemC library [12] a simple FIFO, a pipe, an RSA algorithm and a simple bus implementation have been taken as benchmark. For each design all applicable fault types of the fault model have been evaluated. The designs for the simple FIFO and the RSA algorithm do not have any communication between modules with signals. For this reason, signal binding faults could not be tested. Each type of fault has been injected randomly on three different positions in a design and for each faulty version of the design five traces leading to erroneous output have been applied. Each trace has been initialized with a different assignment of input variables. In general, increasing the number of traces leads to a decreasing size of intersections.

Table 2.1 shows the benchmarks used. In column *LOC* the *lines of code* of the investigated designs are listed excluding the comments. The percentage of the obtained intersected sets of fault candidates is calculated in relation to the size of the design. Also the size of the minimal and the maximal trace are denoted in percent. The percentage of control statements roughly indicates whether the design is control flow or data flow dominated.

For the first two benchmark designs (`simple_fifo` and `pipe`), on average a fourth of the designs has to be analyzed for detecting the faulty statement. In the third benchmark (`rsa`), the percentage of control statements increased and coincidently the average size of the traces decreased compared to the size of the design. However, the blocks that are surrounded by control statements are relatively small while large sequences without any control operations exist. For this reason, the reduction of the traces by intersection is not as significant as for the `simple_bus` benchmark. Although the `simple_bus` is larger, the number of statements in the intersections averages to 6.9%.

Table 2.1 Benchmark designs

Design	Description	*LOC*	Control statements (%)	Intersected trace(%)		
				Min	Max	∅
simple_fifo	Simple FIFO	120	0.5	21.7	26.7	25.0
pipe	Pipeline	220	1.3	24.5	25.5	25.0
rsa	RSA cipher	480	6.5	19.8	24.0	21.3
simple_bus	Abstract bus model	1240	6.6	6.4	8.2	6.9

Table 2.2 Average distance of fault candidates

Fault	simple_fifo	pipe	rsa	simple_bus
Assignment	51.8	27.4	128.7	95.5
Data operation	48.7	24.1	162.4	103.8
Control operation	43.2	21.5	145.9	96.7
Data/port type	53.0	26.5	151.7	109.8
Extra code	37.9	23.4	132.3	99.5
Misplaced code	46.8	23.6	173.5	98.4
Signal binding	–	26.8	–	100.2

2.5.1 Limitations of Quantitative Analysis

Table 2.2 shows the average distance of the fault candidates to the faulty statement in lines of code as explained in Sect. 2.3. Only traces have been considered where the faulty statement is within a trace such that a measurement is possible. Therefore, distance measurements to missing code are not considered.

The experimental results show that the distance strongly depends on the structure of the investigated design and the place where a fault has been injected. For instance, the fault candidates for the simple_fifo benchmark often have a large distance to the faulty statement because the design is mainly sequential where code is carried out successively. The same observation holds for the rsa benchmark. In contrast, the average distance of fault candidates for pipe is relatively small because the pipeline is partitioned into several small functions. For the same reason the distance of the fault candidates to a faulty statement is moderate compared to the size of the SystemC description of the simple_bus. The average distance of the fault candidates is decreasing if a control operation fault is injected such that often the following control-block is carried out. Thus, quantitative analysis significantly varies with the benchmarks.

2.5.2 Qualitative Assessment

In Table 2.3 the applicability of the debugging procedure is evaluated. Column *detection* denotes whether the algorithm is able to detect the specified fault or not. In all cases all types of faults with a checkmark are detectable. This means that the procedure creates a trace which contains the faulty statement causing an unexpected behavior. Vice versa, the other types of faults are not detectable with regard to any trace. Obvious is that the simulation-based algorithm has (expected) weaknesses in localization of design faults. Missing code or missing signal bindings are not detectable because there are no executed statements that are faulty, i.e. there are no faulty statements in the trace.

Table 2.3 Evaluation of simulation-based debugging

Fault	Detection
Assignment fault	✓
Data operation fault	✓
Control operation fault	✓
Data/port type fault	✓
Missing simple code	x
Missing complex code	x
Extra code	✓
Misplaced code	✓
Incorrect/interchanged signal binding	✓
Missing signal binding	x

2.6   Conclusion

Debugging is a process of localization and correction of faults in designs. The problem of evaluating debugging methods has been studied in this paper and a fault model has been proposed that is suitable to analyze the applicability of debugging algorithms. Each type of fault is linked to certain mutations of the PDG. The fault model presented in this paper is extensible and generalizable to other high-level description languages.

A debugging algorithm has been implemented and evaluated with respect to the fault model. The results of the quantitative analysis strongly depend on the structure of the investigated designs. The qualitative analysis has shown that the algorithm is well applicable to detect programming faults while it has weaknesses in detecting certain design faults.

In further work, additional algorithms will be evaluated and compared. Also quantitative approaches that are less dependent on individual benchmarks will be addressed by taking the structure of the source code into account.

References

1. Abadir, M., Ferguson, J., Kirkland, T.: Logic design verification via test generation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **7**(1), 138–148 (1988)
2. Bolchini, C., Miele, A., Sciuto, D.: Fault Models and Injection Strategies in SystemC Specifications. In: EUROMICRO Conference on Digital System Design Architectures, Methods and Tools, pp. 88–95 (2008)
3. Fey, G., Safarpour, S., Veneris, A., Drechsler, R.: On the relation between simulation-based and SAT-based diagnosis. In: Design, Automation and Test in Europe Conference and Exhibition, pp. 1139–1144 (2006)
4. Fey, G., Staber, S., Bloem, R., Drechsler, R.: Automatic fault localization for property checking. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **27**(6), 1138–1149 (2008)

5. Fin, A., Fummi, F., Pravadelli, G.: SystemC as a complete design and validation environment. In: SystemC: Methodologies and Applications, pp. 127–156. Kluwer Academic Publishers (2003)
6. Gallagher, K.B., Lyle, J.R.: Using program slicing in software maintenance. IEEE Transactions on Software Engineering **17**, 751–761 (1991)
7. Hayes, J.: Fault Modeling for Digital MOS Integrated Circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **3**(3), 200–208 (1984)
8. Huang, S.Y., Cheng, K.T.: Formal Equivalence Checking and Design Debugging. Kluwer Academic Publishers (1998)
9. Khalil, M., Traon, Y.L., Robach, C.: Towards an automatic diagnosis for high-level design validation. In: Proceedings of the International Test Conference, pp. 1010–1018 (1998)
10. Köb, D., Wotawa, F.: A comparison of fault explanation and localization. In: International Workshop on Principles of Diagnosis, pp. 157–162 (2005)
11. Renieres, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: Proceedings of the IEEE International Conference on Automated Software Engineering, pp. 30–39 (2003)
12. The Open SystemC Initiative: http://www.systemc.org (2010)
13. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering **28**(2), 183–200 (2002)
14. Zhang, X., He, H., Gupta, N., Gupta, R.: Experimental evaluation of using dynamic slices for fault location. In: Proceedings of the International Symposium on Automated Analysis-Driven Debugging, pp. 33–42. ACM (2005)