ASCHyRO: Automatic Fault Localization of SystemC HLS Designs Using a Hybrid Accurate Rank Ordering Technique

Mehran Goli[1,2] Alireza Mahzoon[2] Rolf Drechsler[1,2]
[1]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[2]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
{mehran, mahzoon, drechsler}@uni-bremen.de

***Abstract*—In order to meet time-to-market constraints and to raise the design productivity, *High-level Synthesis* (HLS) is being increasingly adopted by the semiconductor industry. HLS designs, which can be automatically translated into *Register Transfer Level* (RTL), are typically written in SystemC at the *Electronic System Level* (ESL). However, this modern design flow still has weaknesses, in particular, due to the significant manual effort involved for verification and the subsequent debugging process which are both time-consuming and error-prone.**

In this paper, we propose ASCHyRO, a fully automated semi-formal fault localization approach for SystemC HLS designs. ASCHyRO takes advantage of a hybrid rank ordering technique to derive a reduced ordered set of potential fault locations. The reduced order set is obtained by calculating a *Confidence Score* (CS) for each fault candidate based on a combination of static and dynamic fault probability analysis. Experimental results including an extensive set of standard SystemC HLS designs show the effectiveness of our approach in localizing even multiple faults with high confidence in a short execution time.

I. Introduction

The increasing functionality of digital systems and reduced time-to-market constraints push designers to model systems at a higher level of abstraction than the traditional *Register Transfer Level* (RTL). Hence, *High-level Synthesis* (HLS) at the *Electronic System Level* (ESL) [1] is being increasingly adopted by the semiconductor industry to boost the design productivity [2]. HLS designs are typically developed using SystemC language (a de-facto standard at the ESL) [3], [4] and can be automatically synthesized into RTL. However, this modern design flow still has weaknesses, in particular, due to the significant manual effort involved for verification and the subsequent debugging process which are both time-consuming and error-prone.

SystemC designs verification and debugging are critical, as undetected fault may propagate to final silicon implementation and become very costly to fix. Thus, catching faults as early as possible is of the utmost importance. Although lots of progress has been made in terms of correctness assurance (verification) of SystemC designs [5]–[9], providing an automated debugging solution has received less attention. The debugging process of SystemC designs is a non-trivial task, mainly due to its object-oriented nature and the complex language constructs and semantics (e.g. event-driven simulation semantics, process synchronization, and inherent concurrency). As a consequence, the debugging process of SystemC designs is mostly performed manually that is very time-consuming, making it a bottleneck in the design flow.

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SATiSFy under contract no. 16KIS0821K, and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

In general, the debugging process consists of two main steps which are 1) fault localization, i.e., the identification of possible faulty locations that can cause erroneous state transitions which eventually lead to design failures and 2) fault correction, i.e., locally modifying the functionality of the identified portion. The fault localization is considered as the most time-consuming step in the debugging process and its quality affects the following (manual or automatic) fault correction step [10]. Particularly, localization of functional faults (e.g. false state transition, incorrect assignment, and incorrect operator) is very challenging as they can occur on numerous locations such as local variables, module ports, global signals, conditions, and loops.

In this paper, we focus on the fault localization of SystemC HLS designs that is a critical step of the debugging process at the ESL and typically takes a significant amount of time and effort from designers. We present ASCHyRO, an **A**utomatic Fault Localization of **S**ystem**C** HLS Designs Using a **Hy**brid Accurate **R**ank **O**rdering Technique. ASCHyRO is a semi-formal fault localization approach which consists of two main phases: 1) hybrid program slicing and 2) ranking analysis.

In the first phase, we take advantage of the static and dynamic program slicing methods to confine the initial search space to those parts that actually cause the erroneous output(s). The static slicing is performed on the formal representation of a given SystemC design's behavior i.e., a correlation graph, representing the program based on variables dependency and 2) an adapted model of hammock graph w.r.t SystemC constructs, including a one-to-one correspondence between the design's statements (both data and control). The dynamic slicing is performed on the statements of the design that activate the potential faults at run-time and propagate them to the design's output(s).

In the second phase, results of the static and dynamic slicing are combined to create 1) a *Confidence Score* (CS) for each variable of the design and 2) a reduced set of lines of code (i.e. fault location candidates). The CS is calculated based on two parameters which are the probability of fault occurrence on SystemC design variables derived from the static and dynamic slicing. Variables with the equal CS are categorized in the same class of priority and classes are sorted in descending order based on the CS value. The fault location candidates are obtained based on dynamic slicing of the hammock graph w.r.t the CS classes.

The experimental results, including several standard SystemC HLS designs, demonstrate that ASCHyRO can accurately localize the potential faults in a short execution time and effectively reduce designers effort during debugging process.

```
1  struct M1 : sc_module{
2   sc_in<int> in1,in2,in3;
3   sc_in_clk  clk;
4   sc_in<bool>  ctl1,ctl2;
5   sc_out<int>  out1,out2;
6   sc_out<bool>  ctlOut;
7   void process1();
8   SC_CTOR (M1) {
9    SC_METHOD (process1);
10   dont_initialize();
11   sensitive<<clk.pos();};
12 //-------------------
13 struct M2 : sc_module{
14   sc_in_clk clk;
15   sc_in<int> in1,in2;
16   sc_in<int> ctlIn;
17   sc_out<int>
          out1,out2,out3;
18   int genA,genB,genC,genD;
19   void process2();
20 /*...*/};
21 //-------------------
22 void M1::process1(){
23   int tp1, tp2;
24   tp1=in1.read()*in3.read();
25   tp2=in1.read()+in2.read();
26   if (ctl1.read()){
27    out1.write(-tp1);
28    ctlOut.write(1);}
29   else{
30    out1.write(tp1);
31    ctlOut.write(0);}
32   out2.write(tp2);}
33 //-------------------
34 void M2::process2(){
35   int temp1,temp2,temp3;
36   /*...*/
37   genA = genD;//fault:
          genB = genD;
38   if (genA > in1.read())
39    temp1=in1.read()*genA;
40   else
41    temp1=in1.read()+genA;
42   if (ctlIn)
43    genB = 1;
44   temp2=genA*genB;//fault:
          + -> *
45   temp3=genC+in2.read();
46   out1.write(temp2);
47   out2.write(temp1);
48   out3.write(temp3);}
```

Fig. 1: A part of the *2-stage pipe* design's source code.

II. Related Works

There are several works on program debugging at different levels of abstraction, i.e, from gate [11], [12] to algorithmic levels [10], [19] which are related to our proposed approach, thus we discuss them in this section.

In manual fault localization [13], designers run the design with some input tests until a failure is observed. Then, they set breakpoints iteratively, analyze the program status, and backtrack to the error origin using a source-level debugger such as the *GNU debugger* (GDB) [14]. Since this procedure gives poor results, puts lots of effort on designers, and overall is very time-consuming, alternatives have been developed.

In [15], a debugging approach is introduced to accelerate finding bug in the extracted potential error location set of a given design. In order to rank error candidates, a probabilistic confidence score has been suggested. Similarly, [16] presents a formal debugging method based on static slicing which provides designers with a reduced ordered set of potential error locations. However, both methods are only applicable at RTL and do not support SystemC constructs.

The method presented in [10] enables designers to debug software programs implemented in C language. It provides designers with a set of potential error locations based on dynamic program slicing technique. However, as the method is only based on simulation results, it may fail in case of multiple faults since some of them may not be activated by the input tests at run-time. Moreover, the method uses a commercial tool called "FoREnSiC" to perform dynamic program slicing. This limits the availability of the method (as it is not an open-source free tool). The methods in [17]–[19] propose formal debugging of software program described in C language. However, these debugging methods are at the algorithmic level and hence do not support SystemC constructs. Moreover, a missing formal semantics for the SystemC language restricts the application of formal debugging techniques for SystemC designs at the ESL.

The methods in [20], [21] as well as commercial tools [22], [23] introduce an integrated debugging environment for SystemC designs based on computational reflection, enabling designers to interact with platform simulation models. Although the aforementioned methods help designers in understanding various aspects of a SystemC design and monitoring its behavior, they do not provide any fault localization facilities.

In [24], a simulation-based debugging environment for SystemC designs is proposed. It is based on calculating minimal difference between a passing and a failing process schedule using a set of test cases. However, the method only focuses on process scheduling and does not consider functional faults. The method modifies the SystemC scheduler to handle process activations. This may cause compatibility issue for several approaches in parallel and reduces the degree of automation.

To the best of our knowledge, ASCHyRO is the first SystemC HLS fault localization approach at the ESL that takes advantage of a semi-formal technique to provide designers with a reduced ordered of potential fault candidates.

III. Fault Localization in SystemC HLS

In this section, using a motivating example, we explain challenges of fault localization in SystemC HLS designs and the importance of a proper fault localization approach. Consider the *2-stage pipe* design (Fig. 1) implemented in SystemC. The design includes two modules *M1*, and *M2*, and performs a set of algebraic operations to generate the final results (*out1*, *out2*, and *out3* of module *M2*) in two steps. Here we also assume that the only reference that is available for designers is the correct value of final outputs (as reference results) for a specific testbench. Now consider the scenario that designers made two mistakes when implementing the design. First, an *incorrect assignment* where the definition of variable *genB* of module *M2* (Line 37, Fig. 1) is incorrectly implemented. Second, an *incorrect operator* where the local variable *temp2* of function *process2* of module *M2* (Line 44, Fig. 1) is incorrectly defined. After executing the design, they found that the value of final outputs *out1* and *out2* of module *M2* is incorrect (w.r.t the reference results). Therefore, an important task is now debugging, i.e. localization of the faults and then fixing them.

The common form of debugging is to find a set of potential candidates for the location of the faults. Then, all candidates are evaluated and changes are applied to the design to finally find the exact location of the faults and fix them. A strong fault localization approach significantly reduces not only the required time of the whole debugging process but also effort of the fixing phase. However, fault localization of SystemC designs is very challenging due to the following reasons:

1) The C++ compiler cannot detect the aforementioned types of faults as they are functional faults and are not related to the C++ or SystemC syntax,
2) Monitoring the simulation behavior of the design (e.g. using VCD) [25]–[27] is very time-consuming as it requires to manually trace run-time values of all design's variables. Thus, it does not reduce the number of fault candidates and the search space.
3) The existing C-based debugger methods such as [10], [18] are not applicable as they do not support SystemC constructs, data types and semantics.
4) Since no reference model or a (basic) specification of the motivating example is at hand, the formal methods such as [16], [18] (even if they are adopted to support SystemC

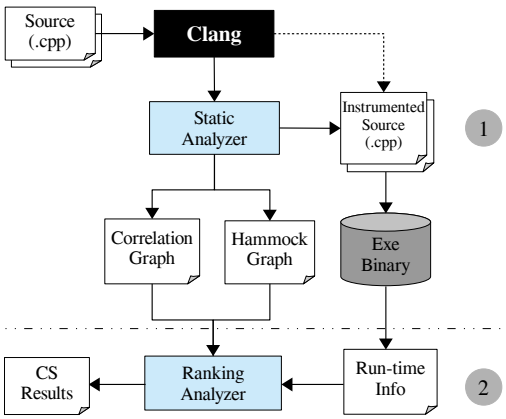

Fig. 2: Overview of the proposed approach ASCHyRO.

constructs) are not applicable. Moreover, as SystemC lacks formal semantics, translating SystemC source code into a formal language is not a trivial task.

In the absence of a proper fault localization approach, the potential fault candidates (or search space) extend to the whole variables of the design and the corresponding lines of code. For example, in the case of motivating example (Fig. 1), all variables and the corresponding lines of code of the design are fault candidates which are 21 variables and more that 46 lines of code (as only a part of the design is represented). However, evaluating and changing all these candidates one by one in order to find the exact fault location is very time-consuming and requires a huge effort. Hence, a fault localization technique for SystemC HLS designs is required to reduce the number of initial candidates and prioritize them for the evaluation and fixing phase.

IV. Fault Localization Methodology

An overview of our proposed approach ASCHyRO is illustrated in Fig. 2, consisting of two main phases which are *Hybrid Program Slicing* and *Ranking Analysis*.

In the first phase, we perform a static and a dynamic analysis. The static analysis consists of two steps:

1) extracting a *Correlation Graph* (CG) to formally slice the program based on variables dependency,
2) analyzing all statements of the design to generate a *Hammock Graph* (HG) including a one-to-one correspondence between the design's statements (data and control),

In order to know which statements of the design contribute to propagating faults to the final output(s) during the execution time, a dynamic analysis is performed. This is done by generating an instrumented version of the source code and executing it with input tests that propagate faults to the output(s) and cause error. The result of this analysis provides designers with dynamic slicing of the SystemC design.

In the second phase, a ranking analysis is performed to calculate a *Confidence Score* (CS) for each variable of the design based on two parameters:

1) probability of fault occurrence on nodes derived from static slicing and
2) probability of fault occurrence on nodes derived from dynamic slicing.

The objective of our proposed approach is to identify a minimal number of variables (and the corresponding lines of code) in a given SystemC HLS design that are responsible for the design's erroneous behavior. In the following, each phase of ASCHyRO is explained in detail and illustrated using the motivating example, introduced in Section III (Fig. 1).

A. Hybrid Program Slicing

The program slicing technique has been introduced for the first time in the software domains to identify parts of a program which have an impact on a selected set of variables [28]. Parts of the program that do not affect these variables are eliminated and hence a reduced set is obtained. This reduced variable set is called a slice. The program slicing technique is divided into static slicing and dynamic slicing [29]. The static slicing extracts all statements that affect the value of a variable for all possible inputs at the point of interest, e.g., at a statement in the program. The dynamic slicing finds those statements that affect the value of a variable for a particular set of input tests applied to the program.

In this paper, we take advantage of a hybrid slicing analysis (combining both static and dynamic techniques) and adopt it 1) to support the SystemC constructs and 2) to be used for fault localization purpose. The goal of static slicing analysis is to formally identify the dependency of a given output to the design's variables for all possible input tests (independent from the testcases). Thus, for the debugging task, it is still valid when new counterexamples are applied to the design. However, the size of the generated slice might be large which tends to a reduction on the impact of the slice on debugging. On the other hand, dynamic slicing can identify those variables (and the corresponding lines of code) that associate to the faulty output for a specific input test. Although the generated slice is more accurate and narrow, it may not be valid when new counterexamples are applied to the design. Hence, the main reason to use a hybrid analysis is that the combination of both techniques not only can reduce the number of fault candidates but also provide the possible fault candidates that are not activated by the given input tests during the execution.

1) Static Slicing: In order to formally represent the behavior of a given SystemC design, we consider two data structures which are CG and HG. The CG data structure describe the behavior of a given SystemC design based on how different variables (including all modules signals, ports and global and local variables) of the design are related to each other. The formal definition of CG data structure is as follows:

Definition 1. *A Correlation Graph (CG) is a structure* $(N,\ E,\ Z)$*, where* N *is a set of nodes,* E *is a set of edges, and* $Z \subseteq N$ *is set of output variables. The edge from node* X *to node* Y *shows that* Y *is dependent to* X*.*

We perform a static analysis on the AST of the design to generate the CG data structure. To do this, first all variables of the design are extracted and tokenized by a unique string including the module, function (for local variable), and variable name. The extraction process is performed by visiting relevant nodes in the AST of the design. To extract variables dependency, a recursive analysis is performed on the AST from the point that computational statements are defined. The computational statements are usually defined as assignments in the design or due to due to the SystemC structure, using the *write()* member function of the output ports. If a statement includes a function (or SystemC process) call, we recursively

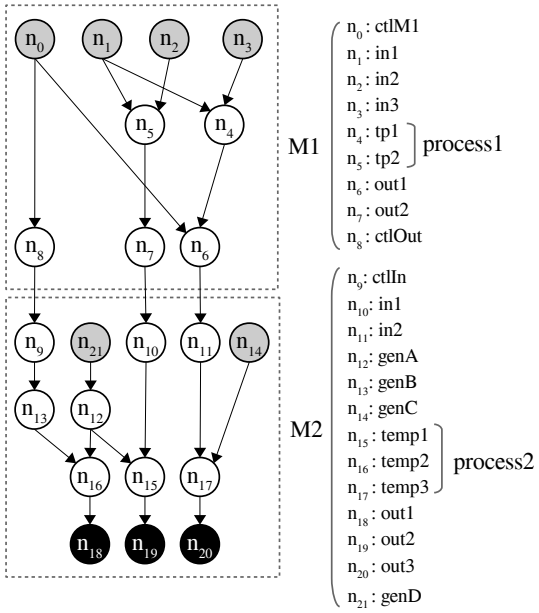

Fig. 3: Coloration Graph (CG) of the motivating example.

extract the relation of the function variables with the left-hand side variable or module port. Moreover, all variables of the compound statements (e.g. *if-else*, *for-loop* or *while* statements) in which these computational statements are defined, are also added into the dependency list of the result variable. It should be taken into account that in our framework, we keep the correspondence between nodes in the correlation graph and variables in the statements of the SystemC code so when a node in the correlation graph is selected as an fault candidate the related statement in the SystemC code is easily determined.

In order to know how different statements (data and control flow) of a given SystemC design are related to each other, the HG representation of the design is automatically generated from the AST of the SystemC design. The formal definition of HG data structure is as follows:

Definition 2. *A Hammock Graph (HG) is a structure* $(N,\ E,\ n_0,\ n_e)$*, where* N *is a set of nodes,* E *is a set of edges in an* $N \times N$ *processing.* n_0 *is the initial node and* n_e *is the end node. If* $(n,\ m) \in E$ *then* n *is an immediate predecessor of* m*, and* m *is an immediate successor of* n*. There is a path from* n_0 *to all other nodes in* N*. From all nodes of* N*, excluding* n_e*, there is a path to* n_e*.*

The HG graph is generated by analyzing the AST of a given SystemC HLS design. We visit all nodes in the AST which are related to statements of the design. This includes both computational and control flow statements. To do this, a *Depth-First Search* (DFS) algorithm is performed within the top level entities (i.e. modules and global functions) to visit all nodes of the statement's type in the AST. We take advantage of modules binding information to extract the connection of modules within the design. Moreover, function calls within the modules process are extracted by visiting the relevant nodes in the AST to understand how lower hierarchies in a module (i.e., local function and process) are connected to each other. Please note that each statement of the design is tokenized by the line of code where the statement is defined.

After extracting both CG and HG, the main task is to identify the nodes located on the cone of faulty output(s), and then slicing the aforementioned graphs into some regions based on the cones. To perform the static slicing, we use definitions of cone and region as follows:

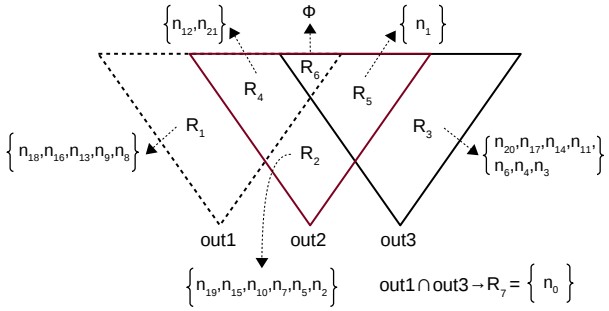

Fig. 4: Region representation of the motivating example's CG.

Definition 3. *In a correlation (or hammock) graph, the cone of output* O_i *is the set of all nodes which* O_i *is dependent on. This cone can be extracted by backtracking the nodes starting from* O_i *and ending in primary inputs.*

Definition 4. *In a correlation graph, assume that* $C_1, C_2, \ldots, C_n$ *are the cones for outputs* $O_1, O_2, \ldots, O_n$*, respectively. The set of nodes which are* ***only*** *shared in cones* $C_{t_1}, C_{t_2}, \ldots, C_{t_k}$ *is called a Region. We show a region based on the dependent outputs as* $R(\{O_{t_1}, O_{t_2}, \ldots, O_{t_k}\})$ *where* O_{t_i} *is the corespondent output for the cone* C_{t_i}*.*

Based on the Definition 4, the identified regions in a CG are completely independent from each other, and each node of the graph is only member of one of the regions. For example, Fig. 3 shows the generated CG of the motivating example (Fig. 1). In this graph, the nodes without any input arrows (the gray nodes) show the primary inputs while the nodes without any output arrows (the black nodes) indicate the primary outputs of the design. The static slicing of the CG based on Definition 3 and Definition 4 is illustrated in Fig. 4. Since the design has three outputs, three cones C_1, C_2, and C_3 are created. $C_1 = \{n_0, n_8, n_9, n_{12}, n_{13}, n_{16}, n_{18}, n_{21}\}$, $C_2 = \{n_1, n_2, n_5, n_7, n_{10}, n_{12}, n_{15}, n_{19}, n_{21}\}$, and $C_3 = \{n_0, n_1, n_3, n_4, n_6, n_{11}, n_{14}, n_{17}, n_{20}\}$ include the set of all nodes on which *out1*, *out2*, and *out3* are dependent, respectively. The regions are created based on Definition 4:

$$R_1(\{out1\}) = \{n_8, n_9, n_{13}, n_{16}, n_{18}\},\quad R_2(\{out2\}) = \{n_2, n_5, n_7, n_{10}, n_{15}, n_{19}\},\quad R_3(\{out3\}) = \{n_3, n_4, n_6, n_{11}, n_{14}, n_{17}, n_{20}\},$$
$$R_4(\{out1, out2\}) = \{n_{12}, n_{21}\},\quad R_5(\{out2, out3\}) = \{n_1\},$$
$$R_6(\{out1, out2, out3\}) = \emptyset,\quad R_7(\{out1, out3\}) = \{n_0\} \qquad (1)$$

The static slicing of HG for a set of faulty outputs is performed by identifying all nodes of the graph on which the outputs are dependent. A node in the HG is considered as a dependent node to a given output, if the corresponding statement of this node (in the design's source code) has at least one variable that is located in the cone of the output. Thus, in the case of HG static slicing, a simplified version of Definition 4 is used where only two regions are created. A region includes the dependent nodes to the faulty output(s) and the other consists of independent nodes.

For example, Fig. 5 shows a part of the generated HG of the motivating example (Fig. 1, Line 37 – Line 48). Each node of the HG represents a statement of the design and is

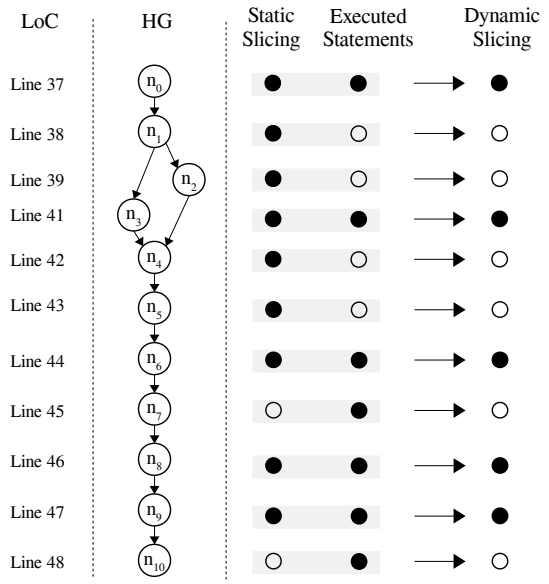

Fig. 5: Static and dynamic slicing of the motivating example based on the generated hammock graph and execution state.

tokenized by the corresponding line of code (LoC). Consider the case of motivating example where the static slicing is performed for *out1* and *out2* of module *M2* (which are two of the design's primary outputs). Black circles in this figure show statements that influence *out1* and *out2*, while white circles show statements that do not have any impact on them. The static slicing of the HG for the primary outputs *out1* and *out2* shows that nodes n_7 and n_{10} do not affect them (as the corresponding statements of these nodes do not have any variables located in the cone of *out1* and *out2*). Thus, the static slicing of the HG for *out1* and *out2* results in the identification of nodes n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_8, and n_9.

2) Dynamic Slicing: In this section, we explain how dynamic slicing for design fault localization is performed and can reduce the search space even more. The dynamic slicing is performed by finding the intersection of the static slices of HG and the executed statements. As we keep the correspondence between nodes in the CG (variables of the design) and nodes in the HG (statements of the design), the dynamic slicing of the CG is performed by extracting the variables that are appeared in the dynamic slicing of the HG. The executed statements are obtained by running the program with the input test(s) that cause faulty output(s). To know which statements (and the corresponding lines of code and variables) of the design have been executed, an instrumented version of the design is automatically generated from the AST. This is done by the *Static Analyzer* module (Fig. 2, phase 1) by adding an instruction after each statement of the design. The generated source code is automatically compiled to an executive binary model and run with the input test(s) that cause incorrect results. Thus, the generated *Run-time Info* includes all executed statements.

Fig. 5 shows an example of the dynamic slicing of a part of the motivating example based on its generated HG and executed statements. The executed statements are obtained for the case that $genA < in1$ (Fig 1, Line 38) and *ctlIn* is equal to zero (Fig 1, Line 42), while other inputs have specific values. Black circles in this figure show statements that influence *out1* and *out2*, while white circles show statements that do not have any impact on them. Due to the input values, the *else* part of the condition statement (Fig 1, Line 41) is only executed, thus nodes n_1 and n_2 are excluded from executed statements. Moreover, nodes n_4 and n_5 are excluded from the executed statements as value of *ctlIn* is zero, i.e., the *if* statement is not executed. By intersecting the result of static slicing of the HG and the executed statements, the result of dynamic slicing of the graph for *out1* and *out2* is generated. It shows the nodes in the HG (the corresponding statements in the source code) that have dependency to *out1* and *out2* of module *M2* and have been executed based on the input tests that cause incorrect results. Thus, the result of dynamic slicing of the HG for *out1* and *out2* consists of nodes n_0, n_3, n_6, n_8, and n_9.

B. Ranking Analysis

The ranking analysis is performed to calculate a CS for each variable of the design and consequently a reduced set of lines of code (i.e. fault location candidates). The CS is obtained using a two-step analysis: static and dynamic ranking.

1) Static Ranking: The goal of static ranking is to specify the exact fault occurrence probability on nodes of CG. The concept of region in Definition 4 empowers us to come up with a theory about the faults location in SystemC HLS designs.

Theorem 1. *Assume that n faults have occurred in the design and propagated to some of the outputs. Assume that $S = \{O_{f_1}, O_{f_2}, \ldots, O_{f_k}\}$ is the set of faulty outputs, and $S_1, S_2, \ldots, S_p$ are all subsets of S. Then, the regions $R(S_{h_1}), R(S_{h_2}), \ldots, R(S_{h_l})$ are the candidates for fault occurrence if $S_{h_1} \cup S_{h_2} \cup \cdots \cup S_{h_l} = S$.*

Proof. Based on the region definition, we know that a region $R(\{O_{t_1}, O_{t_2}, \ldots, O_{t_m}\})$ is on the cone of the outputs $O_{t_1}, O_{t_2}, \ldots, O_{t_m}$. It means that if these outputs are faulty, R is a candidate for the fault occurrence as the fault can propagate to outputs through their cones. Now we can extend the proof for several faults. Assume that we have several faults, a set of regions is a candidate for occurrence of n faults if these regions are located on the cones of all faulty outputs i.e. the regions can influence all faulty outputs. □

In general and with respect to Theorem 1, four *Fault Scenarios* (FSc) might happen when debugging a design. These fault scenarios are illustrated in Fig. 6 and are obtained based on the number of faults n, the number of faulty outputs n_{fout}, and the number of outputs n_{out}. FSc1 (which refers to the case of single fault and single faulty output) and FSc2 (which refers to the case of single fault and multiple faulty outputs) are happened, if the single fault is located in the region of the faulty output(s). Similarly, FSc3 (which refers to the case of multiple faults and single faulty output) is happened, if all faults are located in the reign of the faulty output. Thus, for the aforementioned fault scenarios, the probability analysis is simple as it needs to only be performed on a region.

In contrast, the probability analysis in the case of FSc4 (which refers to the case of multiple faults and multiple faulty outputs) is complex as different regions combination must be taken into account to cover all possible states.

Based on Theorem 1 and w.r.t the fault scenarios, we propose an algorithm to specify the probability of fault occurrence on each node of CG. The algorithm receives a set of faulty outputs, number of faults, and regions on correlation graph

Algorithm 1 Static ranking

Require: Set of faulty outputs $Outs_f$, number of faults n, CG regions $R(Outs_{r_1})$, $R(Outs_{r_2})$, …, $R(Outs_{r_m})$
Ensure: Fault probability for each node p_{n_i}

1: $FaultyRegions \leftarrow \emptyset$
2: $RC \leftarrow$ Create all region combinations with n members
3: **for each** $C \in RC$ **do**
4: $\quad MergedSet \leftarrow \emptyset$
5: $\quad N \leftarrow 1$
6: $\quad$ **for each** $R(Outs_r) \in C$ **do**
7: $\quad\quad MergedSet \leftarrow MergedSet \cup Outs_r$
8: $\quad\quad N \leftarrow N \times size(R(Outs_r))$
9: $\quad$ **if** $MergedSet = Outs_f$ **then**
10: $\quad\quad FaultyRegions \leftarrow FaultyRegions \cup C$
11: $\quad\quad$ **for each** $R(Outs_r) \in C$ **do**
12: $\quad\quad\quad N_{R(Outs_r)} \leftarrow N_{R(Outs_r)} + N$
13: $EffectiveNodeSize \leftarrow 0$
14: **for each** $R(Outs_r) \in FaultyRegions$ **do**
15: $\quad EffectiveNodeSize \leftarrow EffectiveNodeSize + size(R(Outs_r))$
16: **for each** $R(Outs_r) \in FaultyRegions$ **do**
17: $\quad p_{node_{R(Outs_r)}} \leftarrow \frac{R(Outs_r)}{EffectiveNodeSize^n \times size(R(Outs_r))}$
18: $\quad P_{node} \leftarrow P_{node} \cup \{p_{node_{R(Outs_r)}}\}$
19: **return** P_{node}

as inputs and returns the fault probability for each node as output. In order to illustrate each part of the algorithm, we use the motivating example after slicing in Fig. 4 for the FSc4 (which is the most complex fault scenario). With respect to the fault scenario of the motivating example, there are two faults on the correlation graph of Fig. 4 after slicing and both outputs of the design $out1$ and $out2$ are faulty. Therefore, the set of faulty outputs is $S = \{out1, out2\}$. Regions $R_1(\{out1\})$, $R_4(\{out1, out2\})$ are the candidates for the faults occurrence (i.e. one fault on R_1 and one fault on R_4) as $\{out1\} \cup \{out1, out3\} = \{out1, out3\}$ which is equal to the set of faulty outputs S.

Now, we explain our static ranking algorithm in detail with the help of the example. First, all combinations of the CG regions with n members are created (Line 2 in Algorithm 1). As there are n faults in the design, each fault should occur in one of the regions. Thus, the set of region combinations (RC) contains all possible scenarios for occurrence of faults in the design. In our example, the set of RC is as follows:

$$RC = \{\{R_1, R_1\}, \{R_1, R_2\}, \{R_1, R_3\}, \{R_2, R_2\}, \{R_2, R_3\}, \{R_3, R_3\}, \{R_1, R_4\}, \dots\} \quad (2)$$

Next, we analyze each region combination to see if it is a true candidate for the faults occurrence (Line 3). We merge the set of corresponding outputs for the regions in a combination to create $MergedSet$. (Line 7). For instance, as $\{R_1(\{out1\}), R_4(\{out1, out2\})\}$ is one of the combinations in our example, we merge the corresponding outputs to get $\{out1\} \cup \{out1, out2\} = \{out1, out2\}$. Based on Theorem 1, if $MergedSet$ is equal to set of faulty outputs, the region combination is a true candidate for the fault occurrence. Thus, it is added to the set of faulty regions $FaultyRegions$ (Line 9 – Line 10). In our example, $MergedSet = \{out1, out2\}$ exactly equals the set of faulty outputs $Outs_f$, as a result, it is a candidate. Additionally, $\{R_1, R_2\}$, $\{R_1, R_4\}$, $\{R_2, R_4\}$, and $\{R_4, R_4\}$ are other candidates and are identified similarly. We should also calculate the probability of fault occurrence for each combination. Assume that $\{R_{C_1}, R_{C_2}, \dots, R_{C_n}\}$ is a region combination. As these regions are totally independent, the probability of n faults occurrence in this combination is as follows:

$$P = \frac{size(R_{C_1}) \times size(R_{C_2}) \times \dots \times size(R_{C_n})}{size(EffectiveNodes)} \quad (3)$$

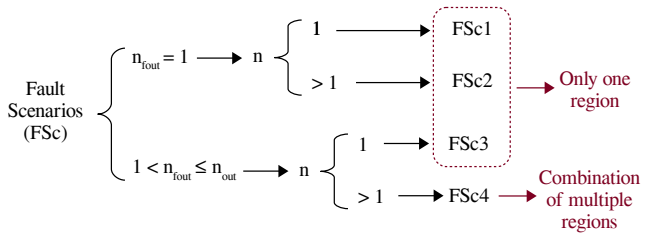

Fig. 6: Possible fault scenarios.

Since the number of effective nodes which have the possibility of fault occurrence is unknown in this point, we only compute the numerator (Line 8). Please note that we do not consider the primary input nodes in the probability computations as it is impossible that a fault occurs on a primary input. Returning to our example again, the values of the numerators for the combinations $\{R_1, R_2\}$, $\{R_1, R_4\}$, $\{R_2, R_4\}$, and $\{R_4, R_4\}$, i.e. all true combination candidates for the fault occurrence are:

$$\begin{aligned} N_{R_1,R_2} &= size(R_1) \times size(R_2) = 5 \times 5 = 25 \\ N_{R_1,R_4} &= size(R_1) \times size(R_4) = 5 \times 1 = 5 \\ N_{R_2,R_4} &= size(R_2) \times size(R_4) = 5 \times 1 = 5 \\ N_{R_4,R_4} &= size(R_4) \times size(R_4) = 1 \times 1 = 1 \end{aligned} \quad (4)$$

In order to obtain the probability of fault occurrence in each single region R_i, we calculate the summation of all combination probabilities where R_i is a member of the combination. Please note that the number of effective nodes is still unknown, we only sum the numerators (Line 11 – Line 12). In our example, the calculations are as follows:

$$\begin{aligned} N_{R_1} &= N_{R_1,R_2} + N_{R_1,R_4} = 25 + 5 = 30 \\ N_{R_2} &= N_{R_1,R_2} + N_{R_2,R_4} = 25 + 5 = 30 \\ N_{R_4} &= N_{R_1,R_4} + N_{R_2,R_4} + N_{R_4,R_4} = 5 + 5 + 1 = 11 \end{aligned} \quad (5)$$

After analyzing all combinations and extracting the complete set of faulty regions $FaultyRegions$, we sum the number of nodes in these regions to obtain the number of effective nodes $EffectiveNodeSize$ (Line 14 – Line 15). Finally, the possibility of fault occurrence for each region is achieved by dividing the numerator value calculated in the previous steps by $EffectiveNodeSize^n$. We also divide the obtained probability by the number of nodes in each region to acquire the probability of the fault occurrence for each CG nodes (Line 16 – Line 19). In our example, The probability of fault occurrence for each node in the three regions are as follows:

$$\begin{aligned} p_{node_{R_1}} &= \frac{N_{R_1}}{EffectiveNodeSize^2 \times size(R_1)} = \frac{30}{11^2 \times 5} = 0.050 \\ p_{node_{R_2}} &= \frac{N_{R_2}}{EffectiveNodeSize^2 \times size(R_2)} = \frac{30}{11^2 \times 5} = 0.050 \\ p_{node_{R_4}} &= \frac{N_{R_4}}{EffectiveNodeSize^2 \times size(R_4)} = \frac{11}{11^2 \times 1} = 0.091 \end{aligned} \quad (6)$$

The probability of fault occurrence for each node is added to the CS of its corresponding variable. Variables with the equal CS are categorized in the same class of priority. A variable (node in CG) with a higher CS should be analyzed earlier in the fixing phase. Therefore, we sort the classes based on their CS value in a descending order.

2) Dynamic Ranking: Although the previous static ranking analysis can greatly reduce the number of fault candidates, there may still be a large number of potential fault locations, especially in the case of complex designs or designs with only

one output. Identifying the true design faults by examining all fault candidates one by one requires a huge amount of run time and effort. To alleviate this problem, we take advantage of the results of dynamic slicing to improve the CS of each design's variables and reduce the number of faulty candidates.

The result of dynamic slicing on CG and HG provide a more reduced set of variables and lines of code than the static slicing, respectively. The goal of dynamic ranking is to specify the exact fault occurrence probability on nodes of CG (variables of the design) which are appeared in the dynamic slicing and improve their CS. Moreover, the corresponding lines of code of each variable w.r.t the dynamic slicing of HG is reported to designers as fault location candidates.

To improve the CS of each variable, a similar probability analysis to Algorithm 1 is performed. The only different is the number of nodes in the CG. The calculated probability value in this step is added to the probability value from static ranking to generate the final CS of each variable.

In our example, after performing dynamic slicing, node n_9 is eliminated from R_1. Thus, the probability of fault occurrence for each node in the three regions are as follows:

$$\begin{aligned} p_{node_{R_1}} &= \frac{N_{R_1}}{\mathit{EffectiveNodeSize}^2 \times \mathit{size}(R_1)} = \frac{24}{10^2 \times 4} = 0.060 \\ p_{node_{R_2}} &= \frac{N_{R_2}}{\mathit{EffectiveNodeSize}^2 \times \mathit{size}(R_2)} = \frac{25}{10^2 \times 5} = 0.050 \\ p_{node_{R_4}} &= \frac{N_{R_3}}{\mathit{EffectiveNodeSize}^2 \times \mathit{size}(R_3)} = \frac{1}{10^2 \times 1} = 0.100 \end{aligned} \quad (7)$$

The final CS for each node (variable of the design) in the CG is obtained by adding the dynamic probability value to the static probability value calculated in the previous step. Therefore, the CS for each node of the CG is as follows: $CS_{n_{12}} = 0.191$, $CS_{n_8,n_{13},n_{16},n_{18}} = 0.110$, $CS_{n_5,n_7,n_{10},n_{15},n_{19}} = 0.100$, $CS_{n_9} = 0.050$, and the CS for other nodes in the CG is zero. By this, we have five classes of variable candidates where the actual faults are in the first two class, including five variables. Moreover, the dynamic slicing of the HG (Fig. 5) is reported to designers to know which lines of code are candidates for fault locations. Thus, in the case of motivating example, the percentage reduction on search space, i.e., variables and lines of code are 76% and 79%, respectively.

V. Experimental Results

The proposed approach was applied to several standard SystemC HLS designs from various domains which are provided by OSCI [30], [31], and [32]. All the experiments were carried out on a PC equipped with 8 GB RAM and an Intel core i7 CPU running at 2.4 GHz. The *Static Analyzer* module was implemented in C++ language using the LibTooling library of Clang compiler [33]. To access relevant nodes in the AST (generated by Clang) of a given design, the primary node visitor *RecursiveASTVisitor* of Clang was used. We used the *Rewriter* interface of Clang to insert the *print* statements in the corresponding design's lines of code and generate its new instrumented version. The *Ranking Analyzer* module was implemented based on Algorithm 1 in C++.

We randomly injected functional faults (such as false state transition, incorrect assignment, and incorrect operator) into each design, which result in faulty (erroneous) output(s). The functional faults that we target in this paper are the common modeling mistakes affecting the functionality of a SystemC design. In order to localize the aforementioned functional faults, we gave the list of buggy output(s), designs' source code, and test cases which only include failing test stimuli to ASCHyRO. To cover all fault scenarios (shown in Fig. 6), we performed 20 experiments for each design (10 for each case of single fault and multiple faults).

Table I shows the results of applying ASCHyRO to different types of SystemC designs for localizing single fault and multiple faults. The first two columns list the names and lines of code for each design, respectively. Column *#Vars* presents the number of variables for each design. Please note that *#LoC* and *#Var* columns also represent the initial search space for fault candidates. Column *#Fault* indicates the number of injected functional faults into each design. The *FLR* shows the percentage of reduction on fault locations w.r.t to the design's lines of code (the initial search space). The *#CFL* column lists the number of code lines as the fault location candidates in the design's source code. Column *VarR* represents the percentage of reduction on number of candidate variables w.r.t to design's variables (the initial search space). The *#CVar* column shows the number of candidate variables for each design which cause erroneous output(s). Please note that the reported reduction values for columns *FLR*, *#CFL*, *VarR*, and *#CVar* are the average of 10 experiments w.r.t single fault and multiple faults.

As can be seen in Table I, ASCHyRO on average reduces the search space by 82% (in terms of LoC) and 78% (in terms of the number of candidate variables), and 76% (in terms of LoC) and 71% (in terms of the number of candidate variables) for single fault and multiple faults, respectively. The worst case reduction results are related to *Cholesky*, *FIR-filter*, *Adpmc*, and *Decimation* benchmarks as they only have one primary output. Thus, the static slicing (and subsequently the static ranking) does not have any effect on reducing the search space as all variables of the design are in the cone of output (i.e. only one region is created). In this case, the reduction only depends on the results of dynamic slicing (and subsequently the dynamic ranking). However, ASCHyRO still can reduce the search space with the help of dynamic slicing. Please note that this limitation is a common problem for all debugging approaches [10], [15], [16] which are based on program slicing and probability analysis. The best case reduction results are related to the *Pkt-switch*, *RISC-CPU*, and *LZW-encoder* benchmarks where up to 96% (in terms of LoC and the number of candidate variables), and 92% (in terms of LoC) and 91% (in terms of the number of candidate variables) reduction in search space were achieved for single fault and multiple faults, respectively. It also shows that our approach can significantly reduce the search space for complex design with a large number of primary outputs.

The execution time of ASCHyRO is illustrated (in seconds) in Table. I, column *Execution Time*, followed by the required time for hybrid program slicing (column *Phase1*), ranking analysis (column *Phase2*) and the total execution time (column *Total*). Column *CET* shows the pure compilation and execution time of each design by GCC without any instrumentation. Please note that the execution time is reported for the average of 10 experiments w.r.t single fault and multiple faults. In comparison to *CET*, the execution time of ASCHyRO is in the same range that on average, it takes 20.39 and 21.95 seconds to localize single fault and multiple faults, respectively. The major time-consuming part of the

TABLE I: Experimental results for all SystemC benchmarks in case of single and multiple faults

	Benchmark	LoC	#Vars	#Fault	#CFL	FLR	#CVar	VarR	Execution Time (s) Phase1	Phase2	Total	CET (s)
Single Fault	4-stage pipe[1]	90	36	1	23	74%	8	78%	6.52	0.11	6.63	3.21
	FIR-filter[2]	365	42	1	91	75%	20	52%	12.32	0.11	12.43	4.11
	Cholesky[2]	432	37	1	137	68%	21	43%	16.39	0.10	16.49	4.38
	Adpcm[2]	444	46	1	129	70%	19	58%	12.51	0.11	12.62	3.78
	Uart[2]	468	55	1	86	81%	6	89%	11.26	0.17	11.43	3.71
	FFT-flpt[1]	586	56	1	77	86%	9	83%	12.28	0.17	12.45	3.92
	FFT-fixed[2]	625	71	1	93	85%	11	85%	14.76	0.19	14.95	4.22
	IDCT[2]	725	64	1	112	84%	6	91%	13.10	0.18	13.28	3.69
	Decimation[2]	793	89	1	219	72%	39	56%	22.97	0.12	23.09	5.02
	VGA[2]	821	81	1	71	91%	4	95%	21.16	0.19	21.35	3.84
	Pkt-switch[1]	1053	72	1	68	93%	5	93%	19.56	0.18	19.74	6.13
	RISC-CPU[1]	1960	345	1	121	94%	13	96%	47.61	0.31	47.92	11.21
	LZW-encoder[3]	5132	422	1	205	96%	21	95%	52.09	0.56	52.65	24.68
	Average	1037.99	108.92	1	110.15	82%	13.99	78%	20.18	0.20	20.39	6.29
Multiple Fault	4-stage pipe[1]	90	36	3	41	54%	13	63%	7.04	0.19	7.23	3.21
	FIR-filter[2]	365	42	2	117	67%	24	44%	12.91	0.12	13.03	4.11
	Cholesky[2]	432	37	3	189	56%	22	39%	17.14	0.11	17.25	4.38
	Adpcm[2]	444	46	2	161	63%	20	55%	12.89	0.12	13.01	3.78
	Uart[2]	468	55	2	93	80%	9	83%	12.73	0.19	12.92	3.71
	FFT-flpt[1]	586	56	4	109	81%	15	72%	13.80	0.20	14.00	3.92
	FFT-fixed[2]	625	71	2	111	82%	13	81%	15.77	0.21	15.98	4.22
	IDCT[2]	725	64	3	129	82%	9	85%	15.09	0.20	15.29	3.69
	Decimation[2]	793	89	2	258	67%	51	42%	23.08	0.13	23.21	5.02
	VGA[2]	821	81	4	97	88%	10	87%	22.48	0.23	22.71	3.84
	Pkt-switch[1]	1053	72	3	82	92%	8	89%	21.61	0.23	21.84	6.13
	RISC-CPU[1]	1960	345	5	290	85%	34	90%	50.36	0.42	50.78	11.21
	LZW-encoder[3]	5132	422	5	410	92%	38	91%	57.29	0.89	58.18	24.68
	Average	1037.99	108.92	3.07	160.53	76%	20.46	71%	21.70	0.25	21.95	6.29

[1]Provided by [30] [2]Provided by [31] [3]Provided by [32] **LoC**: Lines of Code **#CFL**: number of Candidate Fault Location **FLR**: Fault Location Reduction **#CVar**: number of Candidate Variable **VarR**: Variable Reduction **CET**: pure Compilation and Execution Time without any instrumentation

approach is the first phase where the AST of each design and the instrumented version of the source code are generated, CG and HG are extracted, and finally the static and dynamic slicing are performed.

Overall, our experiments with an extensive set of SystemC HLS designs demonstrate that ASCHyRO is efficient and scalable. In particular, even a design with more than $5,000$ lines of code (e.g. *LZW-encoder*) which are the initial search space in debugging process for designers, can be reduced significantly in less than a minute with a high accuracy.

VI. Conclusion

In this paper, we have proposed ASCHyRO, a novel semi-formal fault localization approach for SystemC HLS design at the ESL. Our approach takes advantage of a combination of static and dynamic slicing along with a ranking analysis to localize faults. Our experiment on a wide range of SystemC designs with practical sizes illustrates that ASCHyRO provides designers with a reduced ordered set of fault candidates (variables and the corresponding lines of code) for multiple faults and erroneous outputs in a short execution time.

References

[1] M. Goli and R. Drechsler, *Automated Analysis of Virtual Prototypes at the Electronic System Level: Design Understanding and Applications*. Springer Nature, 2020.

[2] P. Coussy, A. Takach, M. McNamara, and M. Meredith, "An introduction to the SystemC synthesis subset standard," in *CODES+ISSS*, T. Givargis and A. Donlin, Eds. ACM, 2010, pp. 183–184.

[3] "IEEE Standard SystemC Language Reference Manual," 2006, pp. 1–423.

[4] M. Goli, J. Stoppe, and R. Drechsler, "Automated nonintrusive analysis of electronic system level designs," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 39, no. 2, pp. 492–505, 2020.

[5] B. Lin, Z. Yang, K. Cong, and F. Xie, "Generating high coverage tests for SystemC designs using symbolic execution," in *ASP-DAC*, 2016, pp. 166–171.

[6] M. Goli, J. Stoppe, and R. Drechsler, "Automatic protocol compliance checking of SystemC TLM-2.0 simulation behavior using timed automata," in *ICCD*, 2017, pp. 377–384.

[7] ——, "Automatic equivalence checking for SystemC-TLM 2.0 models against their formal specifications," in *DATE*, 2017.

[8] M. Goli and R. Drechsler, "Scalable simulation-based verification of SystemC-based virtual prototypes," in *DSD*, 2019, pp. 522–529.

[9] M. Goli, M. Hassan, D. Große, and R. Drechsler, "Security validation of VP-based SoCs using dynamic information flow tracking," *it Inf. Technol.*, vol. 61, no. 1, pp. 45–58, 2019.

[10] U. Repinski, H. Hantson, M. Jenihhin, J. Raik, R. Ubar, G. D. Guglielmo, G. Pravadelli, and F. Fummi, "Combining dynamic slicing and mutation operators for ESL correction," in *ETS*, 2012, pp. 1–6.

[11] A. Mahzoon, D. Große, C. Scholl, and R. Drechsler, "Towards formal verification of optimized and industrial multipliers," in *DATE*, 2020, pp. 544–549.

[12] A. Mahzoon, D. Große, and R. Drechsler, "RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers," in *DAC*, 2019, pp. 185:1–185:6.

[13] F. Rogin, E. Fehlauer, S. Rülke, S. Ohnewald, and T. Berndt, "Non-intrusive high-level SystemC debugging," in *FDL*. ECSI, 2006, pp. 155–161.

[14] R. Stallman and C. Support, *Debugging with GDB: The GNU Source-level Debugger*. Free Software Foundation, 2010.

[15] T. Jiang, C. J. Liu, and J. Jou, "Accurate rank ordering of error candidates for efficient HDL design debugging," *TCAD*, vol. 28, no. 2, pp. 272–284, 2009.

[16] B. Alizadeh, P. Behnam, and S. Sadeghi-Kohan, "A scalable formal debugging approach with auto-correction capability based on static slicing and dynamic ranking for RTL datapath designs," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1564–1578, 2015.

[17] A. Griesmayer, S. Staber, and R. Bloem, "Fault localization using a model checker," *Softw. Test., Verif. Reliab.*, vol. 20, no. 2, pp. 149–173, 2010.

[18] R. Könighofer and R. Bloem, "Automated error localization and correction for imperative programs," in *FMCAD*. FMCAD Inc., 2011, pp. 91–100.

[19] G. Birch, B. Fischer, and M. R. Poppleton, "Fast model-based fault localisation with test suites," in *TAP*, ser. Lecture Notes in Computer Science, vol. 9154. Springer, 2015, pp. 38–57.

[20] F. Rogin, C. Genz, R. Drechsler, and S. Rülke, "An integrated SystemC debugging environment," in *FDL*, 2007, pp. 140–145.

[21] B. Albertini, S. Rigo, G. Araujo, C. Araujo, E. Barros, and W. Azevedo, "A computational reflection mechanism to support platform debugging in systemc," in *CODES+ISSS*, 2007, pp. 81–86.

[22] "ARM Ltd., MaxSim Developer home," https://www.arm.com, 2006.

[23] "Synopsys System Studio home," https://www.synopsys.com, 2006.

[24] F. Rogin, R. Drechsler, and S. Rülke, "Automatic debugging of system-on-a-chip designs," in *SOCC*, 2009, pp. 333–336.

[25] M. Goli, J. Stoppe, and R. Drechsler, "AIBA: an Automated Intra-cycle Behavioral Analysis for SystemC-based design exploration," in *ICCD*, 2016, pp. 360–363.

[26] M. Goli, M. Hassan, D. Große, and R. Drechsler, "Automated analysis of virtual prototypes at electronic system level," in *GLSVLSI*, 2019, pp. 307–310.

[27] M. Goli and R. Drechsler, "Automated design understanding of SystemC-based virtual prototypes: Data extraction, analysis and visualization," in *ISVLSI*, 2020, pp. 188–193.

[28] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE Trans. Softw. Eng.*, vol. 17, no. 8, pp. 751–761, 1991.

[29] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *SIGPLAN Not.*, vol. 25, no. 6, pp. 246–256, 1990.

[30] A. S. Initiative., http://www.accellera.org/downloads/standards/systemc, 2016.

[31] B. C. Schafer and A. Mahapatra, "S2CBench: Synthesizable SystemC benchmark suite for high-level," *IEEE Embedded Systems Letters*, no. 3, pp. 53–56, 2014.

[32] "Orahyn Ltd., LZW-encoder," https://github.com/arshadri/lzw_systemc/tree/master/systemc, accessed: 2020-01-30.

[33] C. Lattner, "LLVM and Clang: Next generation compiler technology," in *BSD*, 2008, pp. 1–2.