Automated Debugging-Aware Visualization Technique for SystemC HLS Designs

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*—*High-level Synthesis* (HLS) using system-level modeling language SystemC at the *Electronic System Level* (ESL) is being increasingly adopted by the semiconductor industry to raise design productivity. However, errors in the high-level design can propagate down to the low-level implementation and become very costly to fix. Thus, SystemC HLS verification and debugging are necessary and important. While monitoring simulation behavior is a straightforward solution to debug a given design in the case of an error (results of verification), it can become a very time-consuming process as a large amount of data that is not necessarily relevant to the source of error is analyzed.**

In this paper, we propose a fast and automated debugging-aware visualization approach, enabling designers to monitor the portion of a given SystemC HLS design's simulation behavior that is related to the erroneous output(s). Experimental results including an extensive set of standard SystemC HLS designs show the effectiveness of our approach in localizing the designs' simulation behavior in terms of the number of visualized variables. In comparison to traditional visualization methods, our proposed approach obtains up to 96% and 91% reduction in the search space for single and multiple faulty outputs, respectively.

I. Introduction

The emergence of *High-level Synthesis* (HLS) at the *Electronic System Level* (ESL) [1], [2] has significantly reduced time-to-market constraints and boosted design productivity [3]–[5]. HLS designs are being increasingly adopted by the semiconductor industry as an alternative design entry for the traditional *Register Transfer Level* (RTL). HLS designs are typically implemented using SystemC language (a de-facto standard at the ESL) [6] and can be automatically synthesized into RTL. Due to the aforementioned advantages, SystemC HLS designs are used as a reference model for lower levels of abstraction. Hence, verification and debugging of HLS designs are of the utmost importance, as undetected faults may propagate to lower levels of abstraction and become very costly to fix.

The most widely used and scalable SystemC verification approaches [7]–[13] are based on simulation. In this case, the typical simulation-based verification flow consists of stimuli generators where designers simulate a design over a set of concrete test cases and check the final outputs to detect the errors [14]. However, in the case of an error, the subsequent debugging process is still very time-consuming, in particular, as the number of variables and the corresponding lines of code in the design (search space) that must be traced by designers are usually huge. 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., local modification of the identified portion's functionality. The fault localization is considered as the most time-consuming step in the debugging process and its quality affects the following fault correction step [15]. Particularly, localization of timing-based (e.g., incorrect delay) and 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, and global signals.

In the case of an error in one of the outputs of a SystemC HLS design, the straightforward and typical debugging solution is to monitor the simulation behavior of the design and check the run-time values of signals and variables from the point of error (faulty output) back to the primary inputs of the design. However, even for a simple design, using this solution requires a highly manual effort to distinguish the related part of the design code corresponding to the faulty output(s) from non-relevant parts (i.e., fault localization). The lack of analysis or poor results of the first step can put a heavy burden on designers as they have to monitor the whole simulation behavior containing a large amount of data that is not necessarily relevant to the source of error. This can be a very tedious and time-consuming process which makes the HLS debugging a bottleneck in the design flow.

In this paper, we focus on enhancing the SystemC HLS debugging by providing an efficient visualization technique, empowering designers to monitor the detailed simulation behavior of the design related to the erroneous output(s). This can effectively help designers when performing debugging tasks by reducing the initial search space (i.e., all design variables and the corresponding lines of code).

The proposed approach consists of two main phases which are fault localization and localized data visualization. In the first phase, we build on the flexible Clang compiler [16] to statically analyze the *Abstract Syntax Tree* of SystemC HLS designs to extract the structural information, a data dependency graph (representing the design based on variables dependency), and to generate an instrumented version of the design for run-time information extraction. In the second phase, we

```
struct stage1 : sc_module {
 sc_in <bool> clk;
 sc_in <double> din1, din2;
 sc_out <bool> ctl1;
 sc_out <double> dout1, dout2, dout3;
 double varA, varB;
 void prc1();
/*...*/};
struct stage2 : sc_module {
 sc_in<double> din1, din2, din3;
 sc_in<bool> clk;
 sc_in<bool> ctl2;
 sc_out<double> result1, result2;
 void prc2();
/*...*/};
void stage1::prc1(){
 double temp1, temp2, temp3;
 if (varA < varB){
  temp1 = din1.read()*varA + varB;
  ctl1.write(0);}
 else
  temp1 = din1.read() * varA;
 temp2 = varA - varB;
 temp3 = din2.read()*varB;
 dout1.write(temp2);
 dout2.write(temp2 * temp1);
 dout3.write(temp3);}
void stage2::prc2(){
 double sum, mul;
 mul = din1.read() * din1.read();
 sum = din2.read() + din3.read();
 result1.write(sum);
 if (ctl2.read())
  result2.write(mul);
 else
  result2.write(-mul);}
```

Fig. 1: A part of the motivating example source code.

perform a dynamic analysis by executing the instrumented model of the design to extract the run-time value of all design variables and signals. Then, a post-execution analysis is performed to translate the simulation behavior of the design, which is localized with respect to the erroneous output(s), into the *Value Change Dump* (VCD) file. As a result, instead of monitoring and tracing the whole simulation behavior of the design, only a portion of the run-time behavior (including all value changes of local and global variables and signals) is visualized for designers to find the source of error.

The experimental results, including several standard SystemC HLS designs, demonstrate the advantage of our proposed approach in localizing the design's simulation behavior into a subset related to the faulty output(s) in a short execution time which can effectively reduce designers' effort during the debugging process. In comparison to traditional visualization approaches, our proposed approach obtains up to 96% and 91% reduction in the search space for single and multiple faulty outputs, respectively.

II. Related Works

There is a large amount of literature on program debugging and visualization at different levels of abstraction which are related to our proposed approach, thus they are discussed in this section.

The method in [17] proposes a manual SystemC debugging environment which is built on the *GNU debugger* (GDB) [18]. It provides designers with a set of GDB user commands w.r.t SystemC constructs. Thus, designers can set breakpoints iteratively, analyze the program status, and backtrack to the error origin. However, the method does not provide any fault localization mechanism to reduce the search space, meaning the entire debugging process is manual that puts lots of effort on designers, and can be very time-consuming. Moreover, to use the method, the SystemC kernel needs to be modified by designers which may cause compatibility issues for several approaches in parallel and reduces the degree of automation.

In [19], a scalable bug localization tool is presented which relies on combining statistical analysis with HDL slicing. The debugging method in [20] takes advantage of formal techniques such as ranking error candidates where a probabilistic confidence score for each candidate is calculated. Similarly, [21] presents a formal debugging method based on static slicing which provides designers with a reduced ordered set of potential error locations. However, these methods are only applicable at RTL and do not support SystemC constructs.

The method presented in [15] enables designers to debug software programs implemented in C language. It provides designers with a set of potential error locations based on the dynamic program slicing technique. However, it cannot be used to debug SystemC designs as it does not support the SystemC constructs (e.g., module, process, interface, and ports). The methods in [22]–[24] propose formal debugging of software programs described in C language. For example, the method presented in [23] takes an incorrect program and its corresponding specification as inputs and performs symbolic execution and model-based diagnosis for fault localization. However, the aforementioned debugging methods are at the algorithmic level and hence do not support SystemC constructs (e.g., module, process, interface, and ports). Moreover, a missing formal semantics for the SystemC language restricts the application of formal debugging techniques for SystemC designs at the ESL.

There are several SystemC designs visualization environments [25]–[32] as well as commercial tools [33], [34] that help designers to debug a given SystemC design by monitoring its simulation behavior. For example, [25], [35] provide designers with an accurate trace of the simulation behavior of a given SystemC HLS design in the form of VCD. However, the generated VCD file contains the entire simulation behavior of the design (that can be huge) and not the portion that is related to the erroneous output(s). Moreover, as the method takes advantage of GDB to extract run-time information, the analysis time can be enormous especially in the case of complex designs. Although the aforementioned methods help designers in understanding various aspects of a SystemC design and monitoring its behavior, they do not reduce the search space and the number of fault candidates.

In [36], a simulation-based debugging environment for SystemC designs is proposed. It is based on calculating the minimal differences 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 issues for several approaches in parallel and reduces the degree of automation.

The method in [37] proposes a semi-formal fault localization method for SystemC HLS designs. While the results of its analysis can be complementary to our approach, it does not provide designers with any facilities to monitor the simulation behavior of the design which could significantly help them in the fault correction step of the debugging process.

Our proposed approach is fast and automated and targets the entire debugging process where 1) by generating the data dependency graph helps designers in localizing the faults candidates, and 2) by visualizing only the related simulation behavior to the faulty output(s) facilitates the correction step. Moreover, the proposed approach does not rely on any commercial tools.

III. Motivating Example

In this section, using a simple motivating example, we explain the necessity of using an efficient visualization approach for SystemC HLS designs.

Consider the simple SystemC design in Fig. 1 inspired by the standard SystemC design in [38], however, it provides other functionality to support our motivating example. The design includes two modules *stage1* and *stage2*, and performs a set of algebraic operations to generate the final results (*result1* and *result2* of module *stage2*) in two steps. Here also we assume that the only reference that is available for designers is the value of the final outputs (as reference results) for a specific benchmark that they can use to validate and debug the design. Now consider the scenario that designers incorrectly implemented the definition of local variable *temp1* of process *prc1* of module *stage1* (line 19, Fig. 1). After executing the design, they found that the value of the final output *result1* of module *stage2* is incorrect and against the expected output given as the reference results. Since this type of fault is not related to the C++ or SystemC syntax and is a semantic fault, the C++ compiler cannot detect it. Moreover, as neither a reference model nor a (basic) specification of the design is available, monitoring the simulation behavior would be the straightforward solution. This requires to access the detailed behavior of the design, including the run-time values of local and global variables. However, designers can only see the incorrect final output of the design (i.e., *result1* of module *stage2*) and have no information about how other variables associate to generate the final result. Another important point that should be taken into account is that as only one of the final output *result1* is faulty, many variables (and the corresponding lines of code) of the design are not related to it. Therefore, pruning the value changes of those non-relevant variables and signals when visualizing the design simulation behavior can significantly decrease the amount of data that need to be monitored by designers.

In the case of motivating example (Fig. 1), the initial search space includes all variables and the corresponding lines of code of the design which are 19 variables and more than 40 lines of code (as only a part of the design is represented). In the absence of an efficient visualization approach, designers need to either consider the whole simulation behavior of the design to find the source of fault or first manually analyze the source code to realize that they need to only monitor the simulation behavior of 9 variables (out of 19) associated to the faulty output *result1*. However, in both cases, the debugging process can be very time-consuming especially when the design complexity increases. In the following, we show how our debugging-aware visualization approach can enhance the debugging process.

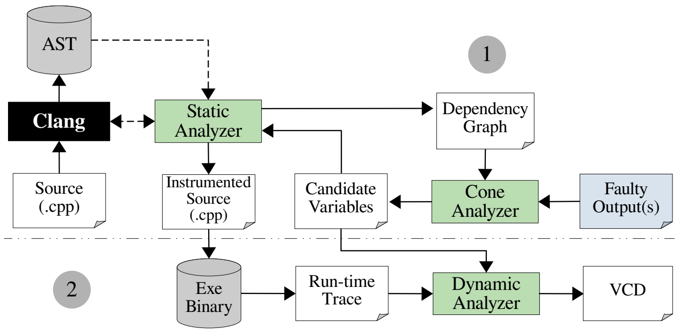


Fig. 2: Overview of the proposed approach.

IV. Debugging-aware Visualization Methodology

A top-level overview of our proposed debugging-aware visualization approach is illustrated in Fig. 2 which consists of two main phases:

1) Fault localization by performing a static analysis on the AST of the design in order to
   - extract the static information to build a data dependency graph,
   - perform cone analysis, and
   - generate an instrumented model of the design for run-time information retrieval.
2) Localized data visualization by performing a dynamic analysis where
   - the instrumented model of the design is executed to extract the run-time value of all design's variables and signals, and

- a post-execution analysis is performed to translate the simulation behavior of the design, which is localized with respect to the erroneous output(s), into the *Value Change Dump* (VCD) file.

In the following, each phase of the proposed approach is explained in detail and illustrated using the motivating example (Fig.1).

A. Fault Localization

The goal of the fault localization phase is to provide a set of variables and signals (and the corresponding lines of code) which is only related to the erroneous output(s).

1) Generating Data Dependency Graph: In order to know how different elements (i.e., variables, modules' ports, and signals) of a given SystemC HLS design are related to each other, we perform a data dependency analysis where for each module of a design, the relation of its (global and local) variables is extracted. The first step of this analysis is to identify and access the node types of the AST which are corresponded to the top-level entities of the SystemC designs (e.g. a SystemC module). Then, we recursively traverse the child nodes of the parent node to reach the other constructs (modules' attributes such as ports or signals) which are defined in the top-level entity. This process is performed for all modules and global functions of the design to generate a data dependency graph w.r.t the following definition.

Definition 1. *The data dependency graph is a structure* (N, E, O)*, where* N *is a set of nodes,* E *is a set of edges, and* $O \subseteq N$ *is set of output variables. The edge from node* n_i *to node* n_j *shows that* n_j *is dependent to* n_i*.*

Each node of the data dependency graph is a variable or signal of the design which is tokenized by the name of module and function (for local variable) to which the variable or signal belongs. For a given SystemC HLS, this graph identifies how the primary inputs and global variables are connected to the final outputs using the intermediate variables and signals. In this graph, the nodes without any input arrows (gray nodes) show the primary inputs or global variables while the nodes without any output arrows (black nodes) indicate the primary outputs of the design. As illustrated in Fig.2 (phase1), *Static Analyzer* module analyzes the generated AST of the design by Clang Compiler and builds up the data dependency graph.

2) Cone Analysis: After generating the data dependency graph, the generated graph and the list of faulty output(s) are sent to *Cone Analyzer* module to identify those nodes located on the cone of the faulty output(s). The cone of output X is a set of all nodes in the data dependency graph on which X is dependent. This cone is extracted by backtracking the nodes starting from X and ending in primary inputs or global variables. By this, the initial search space is pruned and a reduced set of variables are generated w.r.t the faulty output.

In the case of multiple faulty outputs e.g., X1 and X2, a further analysis step is performed by the *Cone Analyzer* module to identify each variable belongs to which cone of the faulty outputs. Therefore, variables are classified in such a way

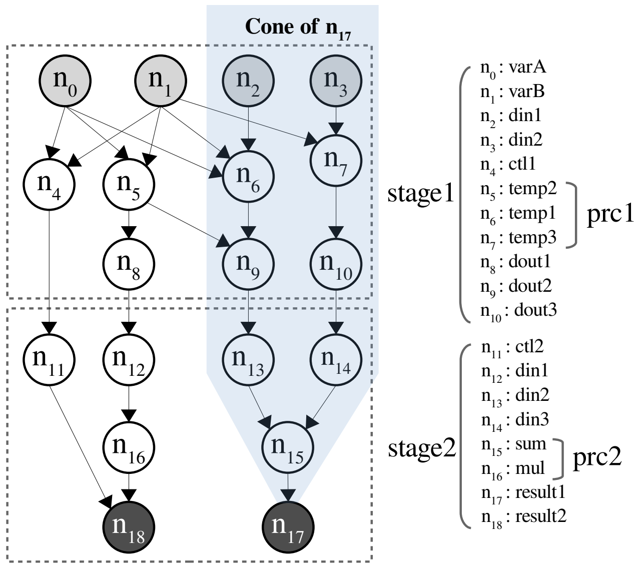


Fig. 3: Part of the generated data dependency graph of the motivating example (Fig. 1).

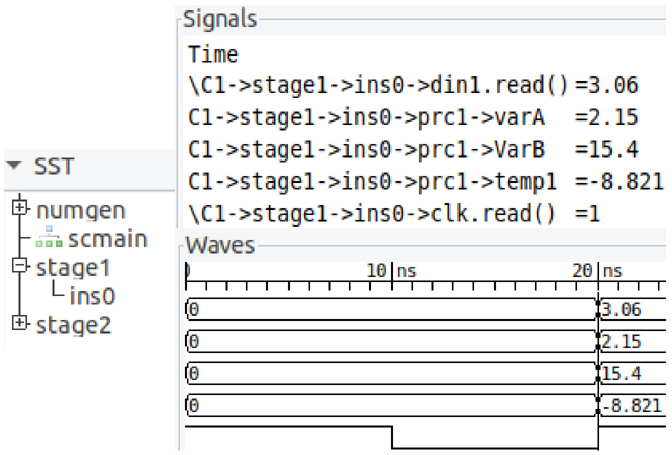
that they either belong to cone X1 (C1) or cone X2 (C2) or the common cone X1 and X2 (C12). We use this information when generating the VCD file of the design to provide designers with a classified representation of the simulation behavior w.r.t the cone of faulty outputs.

For example, the data dependency graph of the motivating example (Fig. 1) is illustrated in Fig. 3. In this figure, nodes n_0, n_1, n_2, and n_3 show the primary inputs while n_{17} and n_{18} indicate the final outputs of the design. Assume that the output signal *result1* of the *stage2* module (node n_{17} in Fig. 3) is faulty. In this case, the *Cone Analyzer* module receives node n_{17} as input and identifies those nodes located *only* on the cone of node n_{17} (C1). These nodes are n_{15}, n_{14}, n_{13}, n_{10}, n_9, n_7, n_6, n_3, and n_2. Please note that nodes n_5, n_1, and n_0 are in the cone of both outputs. Since only one of the outputs is faulty, nodes in the intersection of output cones must be free of bugs. Although it is possible that the bug on a common node is masked for one of the outputs, we assume that there are enough test cases that eliminate this behavior. Thus, w.r.t the initial search space (all variables of the design), a 53% reduction is achieved. Instead of monitoring all variables (nodes) of the design, only a limited number of variables needs to be traced. This can effectively reduce the effort of SystemC designs debugging.

3) Instrumented Code Generation: In this step, results of the *Cone Analyzer* module (stored in *Candidate Variables* file) are used by the *Static Analyzer* module to generate an instrumented version of the design's source code including *Recorder* statements to trace the run-time simulation behavior of the candidate portion in the next phase. The *Recorder* statements are defined based on a hierarchical structure where for tracing a variable, its run-time value, name, the root, and instance name of the module to which the variable belongs

```
1 void stage1::prc1(){
2  double temp1, temp2, temp3;
3  if (varA < varB){
4   temp1 = din1.read()*varA + varB;
5   Fout<<"stage1::prc1::temp1 = "<<temp1<<"
        instance_name_module: "<<this->name()
        <<"simulation_time: "<< sc_time_stamp
        ()<<endl;
6  /*...*/}
```

Fig. 4: A part of the instrumented code of the motivating example (Fig. 1) which is automatically generated.



```
Time
\C1->stage1->ins0->din1.read() =3.06
C1->stage1->ins0->prc1->varA   =2.15
C1->stage1->ins0->prc1->VarB   =15.4
C1->stage1->ins0->prc1->temp1  =-8.821
\C1->stage1->ins0->clk.read()  =1
```

Fig. 5: A part of generated VCD of the motivating example (a screenshot).

are extracted. For local variables of a module's function or process, we extract the name of function or process. This enables designers to trace the variable or signal with a unique identifier that includes hierarchical information about its parent components. The simulation time is also extracted to notify the exact time of the variable's value changes.

In order to accurately trace the run-time value of variables after any changes, the *Recorder* statements are inserted into locations in the source code where

- the variable is defined or declared (e.g. as function, process arguments, or local variables within the function's or process's body),
- the variable is used at the left-hand side of an assignment statement (e.g. expression or function calls), and
- a module port read or write data. This is related to the module ports (e.g. signals with type *sc_in* or *sc_out*) as they use the *read()* or *write()* interfaces to access or modify data.

Coming back to our motivating example (Fig. 1). Assume that we want to trace the local variable *temp3* of the *prc1* process. To do this, the AST of design is analyzed by the *Static Data Analyzer* module to find locations in the source code where the variable is declared, defined, or used (based on the above explanation). For example, consider the location (line 19, in Fig. 1) where variable *temp1* is used in the left-hand side of an assignment expression. The variable is labeled with the name of its parent process (*prc1*) and module (*stage1*). From the extracted information, the *Recorder* statement (Fig. 4, line 5) is automatically generated and inserted after the aforementioned location in the new source code. The instructions *this->name()* and *sc_time_stamp()* are added into the *Recorder* statement to identify that the variable belongs to which instance of module *stage1* and the simulation time when a new value is assigned to the variable, respectively.

B. Localized Data Visualization

By executing the instrumented version of the SystemC design, the run-time value of all variables which are related to the faulty output(s) is extracted and stored in the *Run-time Trace* file. This includes a time-line presentation of all value assignments to every signal and variable w.r.t the program execution order. In order to distinguish variables of different module instances, a unique signature is added to each variable or signal name, including the root and instance name of the module. For local variables of a module's function or process, the name of function or process is also added into the signature of the variable's name. This enables designers to trace a variable or signal with a unique tag that includes hierarchical information about its parent components. We also take advantage of the cone analysis results (see Section IV-A2) to label the variables in the VCD based on the cone of faulty outputs.

As illustrated in Fig. 2 (phase 2), the *Run-time Log* is analyzed by the *Dynamic Analyzer* module to extract the value of all variables at each simulation time stamp and store all information in the *Time-line Log* file. By this, each simulation time stamp is considered as a time unit in which the state of variables that was extracted at this time is stored. One important point that must be taken into account is the presentation of the right value at each time unit for the SystemC primitive data types such as *sc_out* and *sc_signal*. The aforementioned data types have two storage locations for data synchronization. These locations are the *current value* and the *new value*. When a process writes to a signal channel, the process stores the data into the new value rather than into the current value. The value that is stored in the new value location of a signal channel is the right and stable value within a specific delta-cycle. The new value is copied into the current value at the end of the delta-cycle (when the update phase is performed and simulation time advances). Therefore, in a given time unit, the *Dynamic Analyzer* module takes the actual value of SystemC primitive data types from the next time unit (i.e., with one simulation time delay).

After translating the extracted information, the next step is to present this information in such a way that designers can easily trace the states of variables to locate errors in running systems. To visualize the value of SystemC signals, the standard SystemC API (i.e., using *sc_trace*) uses the VCD format for SystemC HLS designs. Although this method works well for SystemC data types which are defined as signals of modules, it comes with several drawbacks as follows.

- It lacks precision for base type variables (e.g. C++ data type) that may change several times during simulation.
- It fails for user-defined datatypes that are not supported at all unless the designers alter their code.
- It cannot trace the values of local variables of modules and functions.
- It requires further programming effort by designers to manually modify the source code and include all signals that need to be traced.

As we take advantage of Clang for the information extraction phase, the extracted run-time data includes all value changes of variables (either global or local or user-defined) during the execution time. Moreover, this process is performed automatically, thus no effort by designers is required.

The generated VCD file includes:

- the cone of faulty output to which the variable belongs, the name of modules, their instances, and the global functions in form of the *Signal Search Tree* (SST),
- the state of the design's variables, and
- the value of each variable, assigned during run-time in the shape of a waveform w.r.t the simulation time.

For example, Fig. 5 shows a part of the generated VCD file of the motivating example where the extracted behavior is shown in three windows: The *SST* window illustrates the name of modules, their instances, and the global functions (e.g. *sc_main* function). The *Signal* window presents the state of the design's variables after the simulation. As an instance, the expression *C1->sage1->ins0->prc1->temp1* shows that the process *prc1* of instance *ins0* of module *stage1* has a local variable *temp1*. This variable is in the cone of faulty output *result1* which is labeled by *C1*. The value of this variable is -8.621 at 20 ns. Finally, the *Waves* window shows the value of each variable that is assigned during run-time w.r.t the simulation time.

With the help of the generated VCD, designers can directly refer to the simulation time when for the first time the primary output *result1* is failed. The data dependency graph not only reduces the search space for designers but also provides them with a backtracking facility. By this, designers can start from the faulty output and trace back through the path specified by the cone of the faulty output and monitors the values of the variables in the VCD to find the source of the bug.

V. Implementation Details

The *Static Analyzer* module is implemented in C++ language using the LibTooling library of Clang compiler [16]. To access relevant nodes in the AST (generated by Clang) of a given design, we use the primary node visitor *RecursiveAST-Visitor* of Clang. It provides the designer with a recursive mechanism on the entire AST to visit each node based on *Depth-First Search* (DFS). The *VisitCXXRecordDecl* (as *SC_MODULE* is defined based on *class* or *struct* in SystemC), *VisitFunctionDecl* and *CXXMethodDecl* are used to find the declaration nodes of modules, functions and SystemC process in the AST, respectively. The information (i.e. name and type) of modules' signals (or ports) and variables is extracted by accessing node's type *FieldDecl*. The local variables of functions are retrieved by visiting node's type *DeclRefExpr*. In order to extract the relation of designs' variables from different statements of the design, the following node types are visited in the AST.

TABLE I: The relationship between different node types in the AST and the corresponding SystemC constructs

AST node type	**SystemC constructs**
CXXMethodDecl	SystemC Process
FieldDecl	SystemC Ports
CXXMemberCallExpr	SystemC Interface (e.g. *sc_out*)
CXXOperatorCallExpr	Function and Process Call
DeclRefExpr	Local and Global Variable
FuncDecl	Local and Global Function
CXXRecordDecl	SystemC Module
IfStmt, ForStmt, WhileStmt	Control Flow

- The *AssignmentOp* node type to retrieve the variables dependency (for both local and global variables) in assignment statements.
- The *CXXMemberCallExpr* and *CXXOperatorCallExpr* node types to find function call and to access modules' ports of types e.g. *sc_in* and *sc_out*, respectively.
- The compound statements node type such as *IfStmt*, *ForStmt*,*WhileStmt* and *SwitchStmt* to extract the dependency of variabes through the control flow of the design (including condition and loop statements).

We use the *Rewriter* interface of Clang to insert the *Recorder* statements in the corresponding design's lines of code and generate its new instrumented version. In summary, the most important Clang constructs (that were used to build the *Static Analyzer* module of the proposed approach) are presented in Table. I. This table also provides a connection between different node types in the AST of SystemC designs and the corresponding SystemC constructs that need to be extracted for our debugging-aware visualization approach. The *Cone Analyzer* and *Dynamic Analyzer* modules are implemented in C++.

VI. Experimental Results

The proposed approach was applied to several standard SystemC HLS designs from various domains which are provided by OSCI [38], [39], and [40]. All the experiments were carried out on a PC equipped with 24 GB RAM and an Intel core i7 CPU running at 1.8 GHz.

We evaluated the quality of the proposed approach in reducing the search space and localizing the design's simulation behavior for two cases of single and multiple faulty outputs. The erroneous output(s) can be caused by different types of faults including functional faults (such as false state transition, incorrect assignment, and incorrect operator), timing-based fault, as well as incorrect locations of variables or instructions in the source code. With the help of the data dependency graph (as it provides localization and backtracking mechanisms) and

TABLE II: Experimental results for all SystemC HLS benchmarks with single and multiple erroneous outputs

Benchmark	LoC	#Vars	#Outputs	Single Faulty Output		Multiple Faulty Outputs		Execution Time (s)			CET (s)
				#VisVar(B–W)	VarR(B–W)	#VisVar(B–W)	VarR(B–W)	Phase1	Phase2	Total	
4-stage pipe[1]	90	36	3	8 – 11	74% – 69%	18 – 29	50% – 19%	5.41	0.11	5.52	3.21
Uart[2]	468	55	3	6 – 22	81% – 60%	31 – 43	43% – 21%	9.76	0.15	9.91	3.71
FFT-flpt[1]	586	56	4	9 – 14	86% – 75%	26 – 39	53% – 30%	10.62	0.16	10.78	3.92
FFT-fixed[2]	625	71	4	9 – 19	87% – 73%	21 – 60	70% – 19%	12.07	0.17	12.23	4.22
IDCT[2]	725	64	4	6 – 15	84% – 76%	19 – 46	70% – 28%	11.49	0.14	11.63	3.69
VGA[2]	821	81	7	4 – 19	91% – 75%	28 – 51	65% – 37%	17.22	0.18	17.04	3.84
Pkt-switch[1]	1053	72	4	5 – 17	93% – 76%	22 – 43	69% – 40%	17.51	0.17	17.68	6.13
RISC-CPU[1]	1960	345	14	13 – 19	96% – 94%	31 – 97	91% – 71%	38.04	0.21	38.25	11.21
LZW-encoder[3]	5132	422	16	21 – 44	96% – 89%	49 – 151	88% – 64%	43.52	0.32	43.84	24.68

[1]Provided by [38] [2]Provided by [39] [3]Provided by [40] **LoC**: Lines of Code **#VisVar**: number of Visualized Variables **VarR**: Variable Reduction
B: Best-case **W**: Worst-case **CET**: Compilation and Execution Time without instrumentation

generated VCD file (as it contains timing information), the proposed approach can facilitate the debugging process of a given SystemC design for all aforementioned fault types.

Table II shows the results of applying the proposed approach to different types of SystemC HLS designs to localize their simulation behavior w.r.t single and multiple faulty outputs. The first two columns list the names and lines of code for each design, respectively. Column *#VisVar* presents the number of candidate variables that are visualized by the proposed approach. The *#VarR* column indicates the percentage of reduction on the number of candidate variables in comparison to the traditional visualization methods such as [25] where the whole design's variables (the initial search space) are visualized. As the search space reduction can be varied based on which output(s) of a given design is faulty, we reported the obtained results for the best (column *B*) and worst (column *W*) reduction cases. Regarding the single faulty output, we assume that each of the design's outputs can be faulty. Thus, we performed the experiment to cover all cases and report the best and worst reduction cases in the table. In the case of multiple faulty outputs, we assume that a given design with N_{out} can have N_{fout} faulty outputs where $1 < N_{fout} \leq N_{out} - 1$. Since performing the experiment to cover all combinations of faulty outputs could be huge in terms of time and state space, we performed up to 10 experiments (w.r.t N_{out}) where we randomly created the faulty output combinations.

As illustrated in Table II, the worst-case reduction results are related to *4-stage pipe*, *Uart*, and *FFT-fixed* benchmarks, especially in the case of multiple faulty outputs. The main reason for the low reduction results is the low number of their primary outputs and also the high dependency of the local and global variables in generating the final results. Thus, in the case of multiple faults, all variables and signals are almost in the cone of the faulty outputs which decreases the impact of cone analysis on reducing the search space. On the other hand, the best-case reduction results (in terms of candidate variables) are related to the *Pkt-switch*, *RISC-CPU*, and *LZW-encoder* benchmarks where up to 96% and 91% reduction in search space were achieved for single fault and multiple faults, respectively. This also shows that our approach can significantly reduce the search space for complex designs with a large number of primary outputs.

Please note that in the case of multiple faulty outputs where the reduction is low, designers can take advantage of the divide and conquer strategy by breaking down the debugging process into smaller steps. With the help of the cone classification of our proposed approach, they can localize the design's simulation behavior for each cone separately. By this, each cone is treated independently results in a significant reduction in the number of variables that are analyzed in each step. Using this strategy, designers can avoid the complex task of analyzing the whole search space at once, and instead they debugging a smaller portion in each step.

The execution time of the proposed approach is illustrated (in seconds) in Table. II, column *Execution Time*, followed by the required time for information extraction and fault localization (column *Phase1*), localized data visualization (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 all experiments w.r.t single fault and multiple faults. In comparison to *CET*, the execution time of the proposed approach is in the same range and in orders of seconds. The major time-consuming part of the approach is the first phase where the AST, the instrumented version of the source code and data dependency graph are generated, and the cone analysis is performed.

Limitation: Although our proposed approach is an overall sound analysis, it comes with a limitation. In the case that a given SystemC HLS design only has one primary output and it becomes faulty, the data dependency graph cannot reduce the search space as all variables of the design are in the cone of faulty output. Please note that this limitation is a common issue for all debugging methods [15], [20], [21], [37] which are based on pruning technique. However, designers can still take advantage of the generated VCD file to accurately trace and monitor the design's simulation behavior.

VII. Conclusion

In this paper, we proposed a fast and automated debugging-aware visualization approach for SystemC HLS design at the ESL. Our approach takes advantage of a combination of static and dynamic analysis to efficiently visualized the portion of a design's simulation behavior that is related to the faulty output(s). Our experiments with an extensive set of SystemC HLS designs demonstrate that the proposed approach is efficient and scalable. In particular, even a design with more than 400 variables and $5,000$ lines of code (e.g. *LZW-encoder*), which are the initial search space in debugging process, can be reduced significantly and visualized in less than a minute with high accuracy.

ACKNOWLEDGMENT

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project VerSys under contract no. 01IW19001, and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

REFERENCES

[1] G. Martin, B. Bailey, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. CA, USA: Morgan Kaufmann Publishers Inc., 2007.

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

[3] P. Coussy, A. Takach, M. McNamara, and M. Meredith, "An introduction to the SystemC synthesis subset standard," in *International Conference on Hardware/Software Codesign and System Synthesis*, 2010, pp. 183–184.

[4] M. Goli and R. Drechsler, "ATLaS: Automatic detection of timing-based information leakage flows for SystemC HLS designs," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2021, pp. 67–72.

[5] ——, "Through the looking glass: Automated design understanding of SystemC-based VPs at the ESL," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021, accepted.

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

[7] A. D. Junior and D. J. C. da Silva, "Code-coverage based test vector generation for systemc designs," in *IEEE Annual Symposium on VLSI*, 2007, pp. 198–206.

[8] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[9] A. Sen and M. S. Abadir, "Coverage metrics for verification of concurrent systemc designs using mutation testing," in *IEEE International High Level Design Validation and Test Workshop*, 2010, pp. 75–81.

[10] B. Lin, Z. Yang, K. Cong, and F. Xie, "Generating high coverage tests for SystemC designs using symbolic execution," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2016, pp. 166–171.

[11] M. Goli and R. Drechsler, "Scalable simulation-based verification of SystemC-based virtual prototypes," in *EUROMICRO Symposium on Digital System Design (DSD)*, 2019, pp. 522–529.

[12] M. Goli, J. Stoppe, and R. Drechsler, "Automatic protocol compliance checking of SystemC TLM-2.0 simulation behavior using timed automata," in *International Conference on Computer Design (ICCD)*, 2017, pp. 377–384.

[13] ——, "Automatic equivalence checking for SystemC-TLM 2.0 models against their formal specifications," in *Design, Automation and Test in Europe (DATE)*, 2017.

[14] B. Khailany, E. Khmer, R. Venkatesan, J. Clemons, J. S. Emer, M. Fojtik, A. Klinefelter, M. Pellauer, N. R. Pinckney, Y. S. Shao, S. Srinath, C. Torng, S. L. Xi, Y. Zhang, and B. Zimmer, "A modular digital VLSI flow for high-productivity soc design," in *Design Automation Conference (DAC)*, 2018, pp. 72:1–72:6.

[15] 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 *European Test Symposium*, 2012, pp. 1–6.

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

[17] F. Rogin, E. Fehlauer, S. Rülke, S. Ohnewald, and T. Berndt, "Non-intrusive high-level SystemC debugging," in *Forum on Specification and Design Languages (FDL)*, 2006, pp. 155–161.

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

[19] M. Jenihhin, A. Tsepurov, V. Tihhomirov, J. Raik, H. Hantson, R. Ubar, G. Bartsch, J. H. M. Escobar, and H. Wuttke, "Automated design error localization in RTL designs," *IEEE Des. Test*, vol. 31, no. 1, pp. 83–92, 2014.

[20] T. Jiang, C. J. Liu, and J. Jou, "Accurate rank ordering of error candidates for efficient HDL design debugging," *IEEE Transactions on Computer Aided Design of Circuits and Systems*, vol. 28, no. 2, pp. 272–284, 2009.

[21] 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 Trans. Computers (TC)*, vol. 64, no. 6, pp. 1564–1578, 2015.

[22] 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.

[23] R. Könighofer and R. Bloem, "Automated error localization and correction for imperative programs," in *Int'l Conf. on Formal Methods in CAD*, 2011, pp. 91–100.

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

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

[26] F. Rogin, C. Genz, R. Drechsler, and S. Rülke, "An integrated SystemC debugging environment," in *Forum on Specification and Design Languages (FDL)*, 2007, pp. 140–145.

[27] B. Albertini, S. Rigo, G. Araujo, C. Araujo, E. Barros, and W. Azevedo, "A computational reflection mechanism to support platform debugging in systemc," in *International Conference on Hardware/Software Codesign and System Synthesis*, 2007, pp. 81–86.

[28] D. Große, R. Drechsler, L. Linhard, and G. Angst, "Efficient automatic visualization of systemc designs," in *Forum on specification and Design Languages (FDL)*, 2003, pp. 646–658.

[29] C. Genz, R. Drechsler, G. Angst, and L. Linhard, "Visualization of SystemC designs," in *International Symposium on Circuits and Systems (ISCAS)*, 2007, pp. 413–416.

[30] P. Pieper, R. Wimmer, G. Angst, and R. Drechsler, "Minimally invasive HW/SW co-debug live visualization on architecture level," in *Great Lakes Symposium on VLSI (GLSVLSI)*, 2021, pp. 321–326.

[31] M. Goli and R. Drechsler, "Automated design understanding of SystemC-based virtual prototypes: Data extraction, analysis and visualization," in *IEEE Annual Symposium on VLSI*, 2020, pp. 188–193.

[32] M. Goli, M. Hassan, D. Große, and R. Drechsler, "Automated analysis of virtual prototypes at electronic system level," in *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, 2019, pp. 307–310.

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

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

[35] 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.

[36] F. Rogin, R. Drechsler, and S. Rülke, "Automatic debugging of system-on-a-chip designs," in *Annual IEEE International SoC Conference (SoCC)*, 2009, pp. 333–336.

[37] M. Goli, A. Mahzoon, and R. Drechsler, "ASCHyRO: Automatic fault localization of SystemC HLS designs using a hybrid accurate rank ordering technique," in *International Conference on Computer Design (ICCD)*, 2020, pp. 179–186.

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

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

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