Exploiting Program Slicing and Instruction Clusterization to Identify the Cause of Faulty Temporal Behaviours at System Level

Moreno Bragaglio, Samuele Germiniani(✉), and Graziano Pravadelli

Department of Computer Science, University of Verona, Verona, Italy
{Moreno.Bragaglio,Samuele.Germiniani,Graziano.Pravadelli}@univr.it

Abstract. Several verification strategies exist to identify unexpected behaviours due to the presence of bugs in system-level HW/SW descriptions. However, when the bug is found, further effort must be spent by the design team to understand its cause and then fix the originating error. This requires a tedious and time-consuming process, generally based on the manual inspection of the execution traces of the design under verification (DUV). This process becomes even more demanding for systems whose behaviours span across wide time windows. Nevertheless, in these cases, usually only a few instructions belonging to long execution traces are relevant for understanding the cause of the unexpected behaviour. Then, we propose a tool that supports the verification engineers in the identification of such a few instructions, to focus their attention on the actual origin of the bug. The tool works by combining dynamic program slicing with a clustering procedure on the execution traces corresponding to unexpected behaviours. Firstly, program slicing is applied to remove instructions not belonging to the cone of influence of the unexpected behaviour. Then, clusters of instructions based on store operations at the LLVM intermediate representation of the DUV are created to guide the heuristic in removing further irrelevant instructions.

Keywords: Bug explanation · Clusterization · Temporal assertions · Program slicing · LLVM · LTL

1 Introduction

Early identification and correction of bugs is a key point in order to save money and speed up the time-to-market of modern embedded systems. In this context, while designers focus on generating a bug-free implementation that meets the specifications, verification engineers work to check that such an implementation indeed satisfies the initial specifications without including unexpected behaviours. Thus, many approaches have been developed both from the point of

© IFIP International Federation for Information Processing 2022
Published by Springer Nature Switzerland AG 2022
V. Grimblatt et al. (Eds.): VLSI-SoC 2021, IFIP AICT 661, pp. 71–92, 2022.
https://doi.org/10.1007/978-3-031-16818-5_4

view of the designers and of the verification engineers to detect bugs and, more generally, unexpected behaviours in system-level descriptions, before they are propagated throughout the lower design levels. However, when such behaviours are found, the verification engineer still has to understand their cause through manual inspection of the execution traces of the design under verification (DUV).

In the context of temporised DUVs, functional requirements involve the concept of time, where behaviours are allowed to span across multiple time units. These behaviours are usually verified using assertions formalised through temporal logic such as linear temporal logic (LTL). Due to its complex nature, understanding and fixing a bug involving temporal logic is way more demanding than finding the cause of an error observable through the failure of a simple propositional assertion. Nonetheless, in both scenarios, understanding the cause of a bug requires a long and tedious manual process of inspection of the execution traces. In most cases, this process is unnecessarily long, since only a few instructions of the execution traces are relevant for understanding and fixing the unwanted behaviours.

To fill in the gap, we present a new methodology and a related tool to automatically remove irrelevant instructions from the execution traces of unexpected temporal behaviours such that verification engineers can focus on the real cause of the problem when debugging their DUV. The tool works on any system-level implementation that can be compiled into a Low-Level Virtual Machine (LLVM) bitcode [1]. Given an unexpected behaviour formalised by means of a propositional assertion, the tool provides the user with a reduced execution trace that still triggers such behaviour, thus highlighting the essential instructions related to it. The underpinning methodology applies a sequence of reductions to the execution trace through a program-slicing-based technique. After each reduction, we verify by simulation if the remaining trace is still an executable program capable of triggering the unexpected behaviour. This procedure works in two phases. Firstly, we remove all the instructions not belonging to the cone of influence of the unexpected behaviour by exploring the dynamic program dependency graph (DPDG). Secondly, we apply a heuristic based on an instruction-clustering procedure to further reduce the remaining trace. In this work, we extend the methodology described in [2] to perform bug explanation of unexpected behaviours modelled as temporal assertions.

The rest of the paper is organised as follows. In Sect. 2, we report the related work. In Sect. 3, we provide a few preliminary definitions necessary to clearly understand the proposed approach. In Sect. 4, we overview the methodology, then we describe in detail each step. In Sect. 5, we describe how to extend the methodology to perform bug explanation with temporal assertions. In Sect. 6, we report the experimental results; finally, in Sect. 7, we draw our conclusions.

2 Related Work

In the last decades, several methodologies, mainly in the software field, have been proposed to tackle the aforementioned problem. A well-known technique to perform fault localisation and bug explanation is, in particular, program slicing.

The original notion of a program slice was proposed by Weiser [3]. Weiser defined a program slice as a reduced program obtained from a program p by removing statements, such that the slice replicates part of the behavior of p. Program slicing techniques fall in two main categories: static and dynamic program slicing. A static slice is computed without making assumptions regarding the input of the program while a dynamic slice relies on some specific test case. Several techniques have been proposed to produce a static slice using reachability algorithms on program dependency graphs (PDG) [4–8]. A PDG is an intermediate program representation to make explicit both data and control dependencies in a program.

Dynamic program slicing was first introduced by Korel and Laski in [9], which allows extracting a (small) executable section of the original program that preserves part of the program's behaviour for a specific input with respect to a subset of selected variables, rather than for all possible computations. One of the most popular applications of dynamic program slicing consists of comparing two or more slices to identify differences or similarities. In [10], the authors present a technique to isolate the region of the bug by computing the difference between a correct slice and the faulty one; likewise, [11] propose an approach to find a correct slice that is the nearest to a related faulty slice. Similar techniques based on intersections and unions between dynamic slices are reported in [12]. In [13], the authors describe a tool to find the cause of a bug by comparing a faulty slice with several correct slices generated through symbolic simulation and converted to sequences of strings. A dynamic program dependency graph is usually employed in conjunction with program slicing as a dynamic variant of a PDG. In a DPDG, dependencies consider a specific occurrence of a certain instruction as there may be several repetitions in a single execution trace. The paper in [14] describes several techniques to exploit a DPDG while performing dynamic program slicing.

Several approaches have been proposed to generate slices by exploiting both static and dynamic information [15–20].

Other approaches rely on statistical methods to perform fault localisation [21,22]. These techniques aim at gathering coverage details of correct and faulty executions over a bugged program, then they rate each programming element in terms of their suspiciousness. In [23] the authors combine dynamic program slicing with statistical methods to build program slicing spectra to rank suspicious elements.

With regard to the use of clustering techniques, Wang et al. [24] proposed a guided technique called "hierarchical program slicing", where the execution trace is divided into phases to simplify the comprehension of data and control dependencies between the instructions in the trace.

The above works provide valid solutions to help the verification engineers in the process of bug localisation and explanation. However, these solutions are usually available only for specific application domains and do not offer a standardised way of defying unexpected behaviors. Furthermore, none of the previous works is capable of providing a reduced execution trace for expected behaviours modelled as temporal assertions.

3 Preliminary Definitions

Definition 1. *An **instruction** is a programming statement following the LLVM bitcode syntax [25].*

Definition 2. *An **execution trace** is a sequence of instructions representing an executable instance of a program.*

Definition 3. *Let i_1 and i_2 be two instructions, i_2 is **data dependent** on i_1 if i_2 accesses a portion of memory allocated or modified by i_1.*

Definition 4. *Let i_1 and i_2 two instructions, where i_1 is a branch with multiple branch targets, if changing the branch target of i_1 may cause i_2 is not executed, then i_2 is **control dependent** on i_1.*

Definition 5. *A **dynamic program dependence graph** is a structure composed of nodes and edges where each node represents an instruction of an execution trace and each edge represents a data or control dependency between instructions. Let n_1,n_2 be two nodes of a DPDG, if n_2 has an incoming edge e_1 connecting n_2 with n_1, then the instruction represented by n_2 is either data dependent or control dependent on the instruction represented by n_1.*

Figure 3 shows an example of a DPDG where red edges are data dependencies and blue edges are control dependencies.

Definition 6. ***Linear temporal logic (LTL)** is a modal temporal logic used to formalise behaviours spanning multiple instants of time. In LTL, one can encode formulae about the future of paths, e.g., a condition will eventually be true, a condition will be true until another fact becomes true, and so on. We recommend [26] for a full reference of the semantics.*

Definition 7. *An **assertion** is a logic property that must hold during the execution of the design. They are divided into two main categories. I) immediate assertion: a function assert defined inside the source code of the design; it checks if a propositional formula is satisfied when assert is called during execution. A proposition can be any kind of Boolean expression that can be constructed in C by connecting variables using boolean, relational or arithmetic operators. II) temporal assertions: a logic formula formalised using LTL. The truth value of the formula is checked by the simulator independently from the execution of the design. In this work, we allow the formalisation of assertions following the grammar in Fig. 1.*

Definition 8. *Let as be an assertion and $A = \{a_0, a_1, ..., a_n\}$ the set of memory addresses of variables $v_0, v_1, ..., v_n$ on which as predicates, then the memory address a^f is a **fundamental address** of as if $a^f \in A$.*

```
template : G(implication)

implication : tformula -> tformula
        | tformula => tformula
        | {sere} |-> tformula
        | {sere} |=> tformula

tformula: proposition
    | (tformula) | !tformula | tformula && tformula
    | tformula || tformula | tformula xor tformula
    | tformula U tformula | tformula W tformula
    | tformula R tformula | tformula M tformula
    | X [N..(N)?] tformula | X tformula
    | F tformula | {sere}

sere : proposition | (sere) | {sere}
     | sere | sere | sere & sere | sere && sere
     | sere;sere | sere:sere | sere[*N(..N)?]
     | sere[*] | sere[+] | sere[=N(..N)?]
     | sere[->N(..N)?] | ##N sere | ##[N(..N)?] sere
     | sere ##N sere | sere ##[N(..N)?] sere
```

Fig. 1. Temporal assertion grammar

4 Methodology

As shown in Fig. 2, the proposed tool is composed of 3 main steps executed sequentially. The inputs of the tool are the LLVM code of the DUV and a set of propositional assertions capturing the expected behaviours. Additionally, the user can provide the sequences of inputs that eventually falsify the assertions, thus highlighting the presence of a bug. For each failed assertion, the tool produces a sequence of minimal instructions explaining the cause of the failure, i.e., the reason for the bug. Hereafter, we provide an overview of the 3 main steps.

1. **Trace Extraction:** given the failure of an assertion, in the first step of the methodology, we extract the sequences of LLVM instructions that brings the execution to activate the unexpected behaviours. This procedure may occur in two ways, depending on whether the user provided the sequences of inputs or only the assertion. In the first case, the sequence of instructions firing the unexpected behaviour is extracted by executing the implementation with the given inputs until the related assertion fails. In the latter case, we use symbolic simulation to find a sequence of instructions capable of falsifying the assertion.

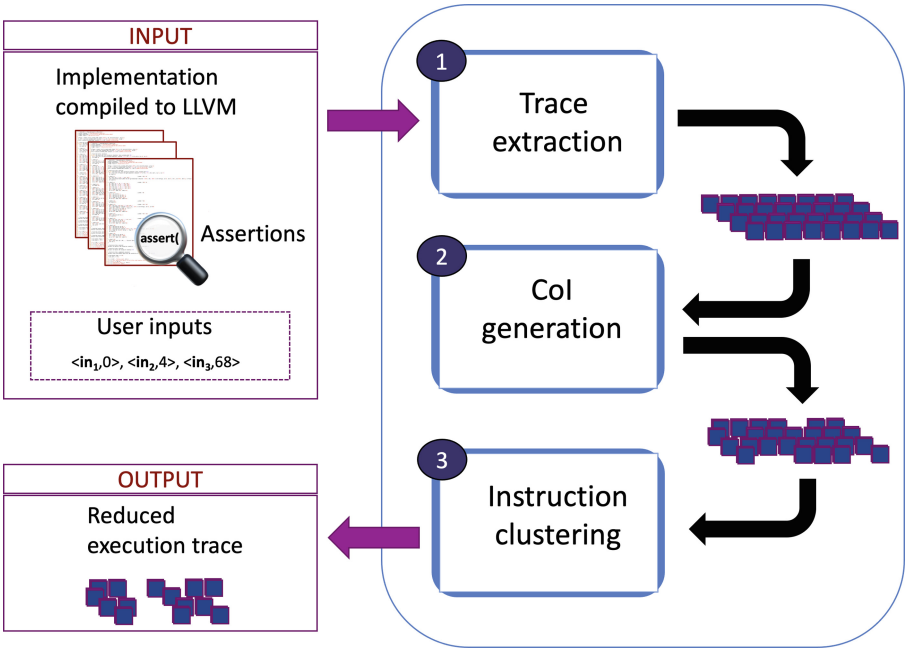

Fig. 2. Methodology execution flow

2. **Cone of Influence Generation:** each trace extracted in the previous step is reduced by applying a dynamic program slicing algorithm to eliminate all instructions not belonging to the cone of influence (CoI) of the assertion. For each trace, we generate a DPDG characterising control and data dependencies between instructions. After that, we apply a reachability algorithm to determine what instructions influence the value of the variables contained in the assertions. The instructions not selected by the above procedure are removed from the trace.
3. **Instruction Clustering:** in the last step of the methodology, we apply a clustering procedure to further reduce the remaining instructions. Our approach consists of dividing the instructions into independent clusters such that applying any reduction procedures to one cluster would not prevent a satisfying minimisation in another. Once such clusters are identified, we apply a combinatorial-based reduction to obtain the minimal sequence in each cluster.

To simplify the exposition, we apply the proposed methodology to the example shown in listing 1.1. It consists of a design written in C implementing a simple arithmetic transformation. The code is decorated with an immediate assertion (line 16) specifying a property that must hold during execution.

```
1  int in;
2  int main() {
3      int a=0;
4      int b=5;
5      while (1) {
6          in= getNextInput();
7          if (in == 0) {
8              a=4;
9              a++;
10         } else if (in < 5) {
11             a+=10;
12             a--;
13         } else if (in > 90) {
14             a-=2;
15             b+=3;
16             assert(a != 12);
17         }
18     }
19 }
```

Listing 1.1. Running example

4.1 Trace Extraction

In the first step of the methodology, we extract the sequences of LLVM instructions that expose the unexpected behaviour, namely, sequences starting with the first instruction of the program and ending with the assertion failure.

In Table 1 we report an execution trace falsifying the assertion contained in the running example. The instructions are labelled with two identifiers: the first uniquely identifies each LLVM instruction, the second links each instruction to its corresponding high-level statement in listing 1.1. To extract such an execution trace, we symbolically simulate the DUV, until we find an execution path that falsifies the target assertion. To accomplish that, we exploit the symbolic simulation engine provided by KLEE [27]. To simulate the DUV with KLEE, the DUV inputs are marked as "symbolic" to declare where symbolic values should be injected. For example, to symbolically simulate the running example, line 6 must be replaced by $klee\_make\_symbolic(in)$, since variable in is the only input. Then the symbolic simulation explores the various paths of the running example, until it finds a path that makes the assertion at line 16 fail. Such a path has the following symbolic constraints: $(in1 == 0, in2 < 5, in3 > 90)$, where the subscript i of ini refers to the value of the variable in at the symbolic iteration i.

Symbolic simulation is quite expensive in terms of computational resources. As a matter of fact, it is an exponential-time algorithm; however, if the user already has the required sequence of inputs to activate the bug, it can be run in a linear-time constrained mode, since only one path needs to be explored. In the running example, we assumed the following sequence of inputs: $\{\langle in1, 0\rangle,$

Table 1. LLVM execution trace of the running example

<label>:0:	**<label>:14:**
[0, 1] %1 = alloca i32	[26, 11] %15 = load i32, i32* %2
[1, 3] %2 = alloca i32	[27, 11] %16 = add add nsw i32 %15,10
[2, 5] %3 = alloca i32	[28, 11] store i32 %16, i32* %2
[3, 1] store i32 0, i32* %1	[29, 12] %17 = load i32, i32* %2
[5, 3] store i32 0, i32* %2	[30, 12] %18 = add nsw i32 %17, -1
[7, 4] store i32 5, i32* %3	[31, 12] store i32 %18, i32* %2
[8, 5] br label %4	[32, 18] br label %31
<label>:4: //in=0	**<label>:31:**
[9, 6] store i32 getNextInput(), i32* %1	[33, 5] br label %4
[10, 7] %6 = load i32, i32* %1	**<label>:4:** //in=125
[11, 7] %7 = icmp eq i32 %6, 0	[34, 6] store i32 getNextInput(), i32* %1
[12, 7] br i1 %7, label %8, label %11	[35, 7] %6 = load i32, i32* %1
<label>:8:	[36, 7] %7 = icmp eq i32 %6, 0
[13, 8] store i32 4, i32* %2	[37, 7] br i1 %7, label %8, label %11
[14, 9] %9 = load i32, i32* %2	**<label>:11:**
[15, 9] %10 = add nsw i32 %9, 1	[38, 10] %12 = load i32, i32* %1
[16, 9] store i32 %10, i32* %2	[39, 10] %13 = icmp slt i32 %12, 5
[17, 18] br label %31	[40, 10] br i1 %13, label %14, label %19
<label>:31:	**<label>:19:**
[18, 5] br label %4	[41, 13] %20 = load i32, i32* %1
<label>:4: //in=4	[42, 13] %21 = icmp sgt i32 %20, 90
[19, 6] store i32 getNextInput(), i32* %1	[43, 13] br i1 %21, label %22, label %31
[20, 7] %6 = load i32, i32* %1	**<label>:22:**
[21, 7] %7 = icmp eq i32 %6, 0	[44, 14] = load i32, i32* %2
[22, 7] br i1 %7, label %8, label %11	[45, 14] %24 = sub nsw i32 %23, 2
<label>:11:	[46, 14] store i32 %24, i32* %2
[23, 10] %12 = load i32, i32* %1	[47, 15] %25 = load i32, i32* %3
[24, 10] %13 = icmp slt i32 %12, 5	[48, 15] %26 = add nsw i32 %25, 3
[25, 10] br i1 %13, label %14, label %19	[49, 15] store i32 %26, i32* %3
	[50, 16] %27 = load i32, i32* %2
	[51, 16] %28 = icmp ne i32 %27, 12
	[52, 16] %29 = zext i1 %28 to i32
	[53, 16] %30 = call @assert

$\langle in2, 4\rangle$, $\langle in3, 125\rangle\}$. Therefore, the symbolic simulation must explore only one path with the following constraints: $in1 == 0$, $in2 == 4$, $in3 == 125$ producing the sequence of instructions reported in Table 1.

4.2 Cone of Influence Generation

In the second step of the methodology, the execution trace extracted in the previous phase is reduced by applying a dynamic program slicing algorithm. The remaining elements of the execution trace correspond to instructions involved directly (or indirectly through association) in data or control dependencies with the variables contained in the failed assertion, that is, the cone of influence of the assertion. The procedure works in three main sub-steps.

In the first step, we generate the DPDG of the execution trace extracted in the first step of the methodology. In the last decades, many algorithms have been proposed to generate DPDGs efficiently, one of which can be found in [28]; therefore, we do not describe such an algorithm in this paper. Figure 3 shows the DPDG for the execution trace listed in Table 1.

In the second step, we identify all *store* instructions in the execution trace accessing fundamental addresses for the target assertion. These are the only instructions that can modify the variables on which the assertion predicates, and therefore, that can change its truth value. We call $fundInst$ the set of instructions collected with the above procedure. Since the algorithm to identify $fundInst$ is trivial, we do not give any further details on it. In the running example, there is only one fundamental address, namely, the memory address of variable a in assertion $a! = 12$. Such an address is allocated by instruction 3 of Table 1 and saved in the LLVM label %2. In this example, $fundInst$ is composed of the store instructions $\{5, 13, 16, 28, 31, 46\}$, which are accessing the address in label %2.

In the last step, we traverse the generated DPDG starting from each store instruction in $fundInst$ and going backward through the incoming edges until a node with no incoming edges is found. By construction, the generated DPDG is an acyclic direct graph, therefore the whole procedure has worst-case time-complexity of $O(V)$, where V is the number of nodes in the DPDG. Each instruction represented by a node in the DPDG that is not visited in the aforementioned procedure will be removed from the execution trace. The whole procedure is formalised in function *extractCoI* of Algorithm 1.

The inputs of this function are the identifiers corresponding to fundamental instructions $fundInst$, the execution trace *trace* and the DPDG *dpdg*. First, *visited* and $reducedTrace$ are declared and initialised (line 2, 3); the first variable contains the visited nodes, while the latter contains the reduced execution trace. After that, we apply the function *backwardDFS* to all the nodes representing the fundamental instructions in $fundInst$ (line 4–6). Each node is retrieved from the DPDG through the method $getNodeFromeId$ (line 5) which returns a node data structure for a given instruction identifier. The function *backwardDFS* performs a depth-first search algorithm going backward from the incoming edges of each node. First, the function marks the current node as visited (line 17). After that, it iterates through all the incoming edges of the current node (line 18). Then, it retrieves the source node *sourceNode* connected to *node* through *edge* using the method *getSource* (line 19). If *sourceNode* is not already marked (line 20), then we apply *backwardDFS* recursively using *sourceNode* as input (line 21).

Algorithm 1. Cone of influence extraction

```
1: function extractCoI(fundInst, trace, dpdg)
2:     visited = ∅
3:     coi_Trace = ∅
4:     for all fi_id in fundInst do
5:         node = dpdg.getNodeFromId(fi_id)
6:         backwardDFS(node, visited)
7:     end for
8:     for id = 0, id < trace.size(), id++ do
9:         if !visited.contains(id) then
10:            reducedTrace.pushBack(trace[id])
11:        end if
12:    end for
13:    return reducedTrace
14: end function
15:
16: function backwardDFS(node, &visited)
17:    visited.insert(node)
18:    for all edge in node.getInEdges() do
19:        sourceNode = edge.getSource()
20:        if !visited.contains(sourceNode.getId()) then
21:            backwardDFS(sourceNode, visited)
22:        end if
23:    end for
24: end function
```

When all the visits are concluded, we iterate on all the instructions in *trace* (line 8) and we add to *coi_Trace* the instructions that do not have a corresponding marked node (line 9–10), that is, that do not have a corresponding node stored in *visited*. Finally, the reduced trace is returned (line 13).

If we apply the above procedure to the running example, the instructions corresponding to nodes 2, 7, 47, 48, 49 are removed from the trace. These nodes are highlighted in red in Fig. 3. Intuitively, these instructions refer to the declaration and utilisation of variable b, which does not have any control or data dependency with variable a in the assertion. From now on, we will use the term *CoI-Trace* to refer to the execution trace reduced with the above procedure.

4.3 Instruction Clustering

In the last step of the methodology, we apply a heuristic procedure to further reduce the remaining instructions in the CoI-Trace. Further reductions are necessary because in most cases, step two of our methodology can not produce a minimal sequence of instructions falsifying an assertion. Consider, for example, the high-level instructions $a{+}{+}$ and $a{-}{-}$ contained, respectively, at lines 9 and 12 of the running example. Since the assertion predicates on variable a, which is data-dependent on these instructions, the previous step is not capable

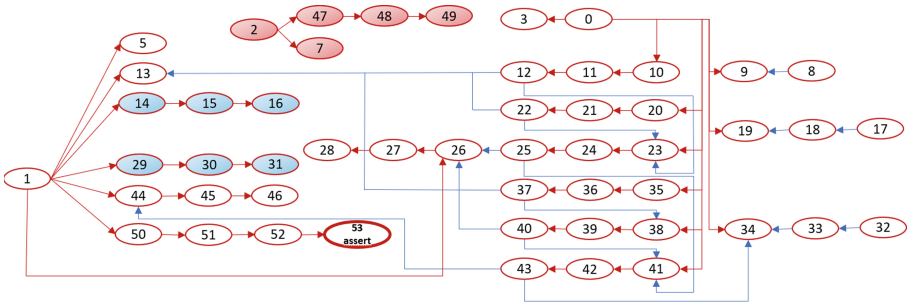

Fig. 3. DPDG of the running example (Color figure online)

of removing them. In theory, any subsequence of instructions of the execution trace could be a minimum sequence of instructions explaining the unexpected behaviour. Therefore, any algorithm seeking to find the minimal sequence would suffer from exponential complexity, and hence, scalability issues. To tackle this problem, our approach splits the instructions of the CoI-Trace into independent clusters such that applying any reduction to one cluster would not prevent a satisfying reduction to another cluster. Since every cluster contains a small number of instructions, it is feasible to quickly find the optimal reduction for each cluster. We generate such clusters by grouping store instructions accessing the same memory address. Note that this is just one method of clustering the instructions, the whole methodology can be still applied with different heuristics. Our clustering heuristic does not produce clusters completely data/control independent from one another; nonetheless, they provide a satisfying amount of independence to apply effective individual reductions. Since each store instruction can only access one memory address, the required clustering procedure is straightforward. In the running example, the clustering procedure produces two clusters for the execution trace of Table 1: $c_1 = \{13, 16, 28, 31, 46\}$ for the stores instructions accessing to the address of variable a, and $c_2 = \{9, 19, 34\}$ for the address of variable in.

Let $a_i, a_2, ..., a_k$ be the addresses accessed in the store instructions of the CoI-Trace, $C = \{c_1, c_2, ..., c_k\}$ is the set of clusters generated with the above procedure, where c_i contains the store instructions for address a_i. We define the **optimal reduction** as the biggest set of instructions $optRed_i = \{i_1, i_2..., i_m\}$ in a cluster c_i such that if the execution trace is stripped of the instructions contained in $optRed_i$, the trace is still an executable program capable of falsifying the assertion. For each cluster, we find its optimal reduction and we remove the respective instructions from the execution trace. In the running example, instructions 16 and 31 correspond to the optimal reduction of cluster c_1. We identify a candidate optimal reduction $optRed_i$ of a cluster c_i by applying a "select and test" procedure. Firstly, we select a subset $s_i \subseteq c_i$, then we remove the selected instructions from the trace. Secondly, we test if the execution trace is still an executable program capable of falsifying the assertion. To perform such

Algorithm 2. Reduction through clustering and slicing

```
1: function reduce(trace,dpdg)
2:     finalTrace = trace
3:     C = generateClusters(trace)
4:     for all c_i in C do
5:         for s = c_i.size(), s > 0, s-- do
6:             combs = getCombs(c_i.size(), s)
7:             for all comb_i in combs do
8:                 c_sel = select(c_i, comb_i)
9:                 trace^s = strip(c_sel, finalTrace)
10:                if test(trace^s) then
11:                    removeLooseInst(c_sel, dpdg, trace^s)
12:                    finalTrace = trace^s
13:                    goto newCluster
14:                end if
15:            end for
16:        end for
17:        label newCluster
18:    end for
19:    return finalTrace
20: end function
21:
22: function removeLooseInst(c_sel,dpdg,&trace^s)
23:    for all c_j in c_sel do
24:        visited = ∅
25:        node = dpdg.getNodeFromId(c_j)
26:        removeLooseNodes(node, visited)
27:        trace^s.erase(visited)
28:    end for
29: end function
30:
31: function findLooseNodes(node,&visited)
32:    if node.getInEdges().size() > 1 then
33:        return
34:    end if
35:    visited.insert(node)
36:    for all edge in node.getInEdges() do
37:        sourceNode = edge.getSource()
38:        findLooseNodes(sourceNode, visited)
39:    end for
40: end function
```

a test, we exploit the KLEE LLVM interpreter to re-execute the reduced trace. This procedure can produce only three outcomes: (1) the assertion fails during execution; (2) the assertion does not fail; (3) a branch instruction jumps to a different target than the one in the original trace.

In the first scenario, removing the instructions does not affect the truth value of the assertion, hence, the removed instructions are considered a candidate opti-

mal reduction. On the contrary, in the second and third scenario, the removed instructions were necessary to, respectively, falsify or reach the assertion, therefore, they can not be removed from the trace. The biggest candidate optimal reduction identified with the above procedure is the optimal reduction for the given cluster. Step three of our methodology is completely formalised in the function *reduce* of Algorithm 2. First, the function generates the clusters of stores instructions (line 3) through the method *generateClusters*. Then, the selection and test procedure is performed for all clusters. The selection phase works by selecting progressively smaller combinations of cluster instructions (lines 4–8). For example, let $c_p = \{23, 45, 98\}$ be a cluster of instructions, the selection phase starts by selecting combinations of size 3, which is only $\langle 23, 45, 98\rangle$. After that, it continues with combinations of size two, which are $\langle 23, 45\rangle$, $\langle 23, 98\rangle$, $\langle 45, 98\rangle$ and finishes with combinations of size 1, which are $\langle 23\rangle$, $\langle 45\rangle$, $\langle 98\rangle$. For each combination, a new reduced trace $trace^s$ is generated by removing the corresponding instructions using function *strip* (line 9). $trace^s$ is re-executed through function *test* (line 10). If *test* returns true, then we are in scenario 1 of the aforementioned procedure and c_{sel} is an optimal reduction of c_i. In this case, the newly reduced trace is saved in $finalTrace$ (line 11) and the execution moves to the next cluster (line 13). Finally, when the trace is reduced using all clusters, we return the final trace (line 19). If we apply this procedure to cluster c_1 and c_2 of the running example, we discover that there is no candidate reduction for c_2 as all its store instructions are necessary to explain the unexpected behaviour; on the contrary, cluster c_1 admits an optimal reduction consisting of instructions 16 and 31.

In most cases, removing a store instruction i_s generates a chain of "loose instructions" i_1, i_2,..., i_{p-1}, i_p where i_s is data dependent only to i_1, i_1 is data dependent only to i_2 ..., i_p is data dependent only on i_{p-1}. Since i_1 is the only data dependence of i_s, removing i_s causes i_1 to become independent from all the other instructions in the trace. Therefore, since i_1 is no longer part of the cone-of-the influence, we can safely remove it from the trace. In the same way, $i_2 ... i_{p-1}$, i_p are removed in a chain-reaction fashion once their only dependence is removed. The above procedure is implemented by the function $removeLooseInst$ of Algorithm 2. The inputs of $removeLooseInst$ are the store instructions c_{sel} removed in the previous iteration of *reduce*, the DPDG *dpdg* and the stripped trace $trace^s$. The procedure works in two phases executed for every instruction in c_{sel} (line 23). First, it finds the nodes *visited* corresponding to loose instructions in *dpdg* using function $findLooseNodes$ (line 24–26). Second, the found instructions are removed from $trace^s$ (line 27). Function $findLooseNodes$ performs the same task of $backwardDFS$, except that it returns when a node with more than one dependence is found (line 32). By removing instructions 16 and 31 in the running example, we generate the loose instructions 14, 15 and 29, 30, respectively. These instructions are removed automatically through the $removeLooseInst$ function. Overall, step three of the methodology removes instructions 14, 15, 16, 29, 30, 31 whose corresponding nodes are highlighted in blue in Fig. 3.

5 Bug Explanation with Temporal Assertions

In this section, we describe how to extend the methodology in Sect. 4 to perform bug explanation where the unexpected behaviour is identified through a failing temporal assertion. First, we describe how to handle the advancement of time (Sect. 5.1). After that, we report how to extract an execution trace that makes a temporal assertion fail (Sect. 5.2). Finally, we show how to modify the extracted trace in order to apply the techniques explained in the second and third steps of the methodology (Sect 5.3).

5.1 Time Flow

Temporal assertions are an invaluable tool to verify synchronous RTL designs where the advancement of time is usually defined through a clock signal. Each time a clock signal reaches a positive (or negative) edge, time advances by 1 unit inside the assertion. However, in the specific domain of application of this work, there is no signal that is responsible for articulating the advancement of time. To solve this issue, in this work time advances by one time unit each whenever a new input is provided to the design. The values of the variables inside an assertion at time t_i (corresponding to the i-th input) are equal to the values of the corresponding variables inside the design before executing the instructions necessary to read $input_{i+1}$. In the running example, the value of variable a is equal to 0 at time t_0, before reading the first input. a becomes equal to 5 after receiving the first input $\langle input_1, 0 \rangle$ at time t_1. Note that inside the assertion, the first evaluation unit is t_1 (first sample of the variables) and not t_0.

If the executions reads multiple consecutively inputs, they are all considered part of the same time unit. For example, if the execution is currently at time t_j and the simulation must execute the following instructions

```
1    in1 = getNextInput1();
2    in2 = getNextInput2();
3    in3 = getNextInput3();
```

then, time is equal to t_{j+1} after executing the third statement. This is necessary to allow the evaluation of multiple inputs on a single time unit.

In this work, we consider only safety assertions following the template $always(antecedent \rightarrow consequence)$ (see Definition 7) where both the antecedent and the consequent can be any LTL temporal formula.

5.2 Trace Extraction

Evaluating temporal assertions while performing symbolic simulation presents several additional issues, we describe the main challenges below.

– The assertion is no longer part of the source code of the design; therefore, it must be handled by the simulator outside the simulation.

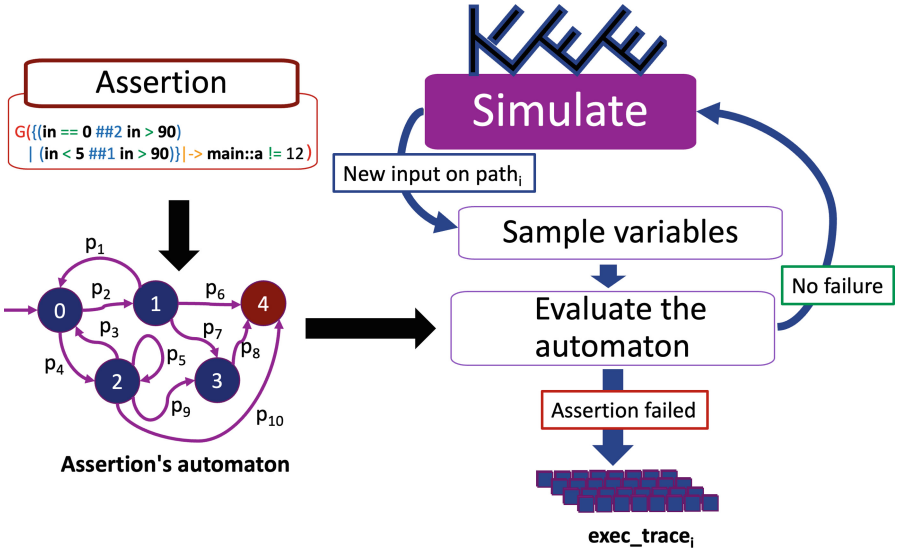

Fig. 4. Trace extraction with temporal assertion

– The variables used inside an assertion might not be always available during simulation; this happens because the existence in memory of a variable depends on the scope in which it is declared.
– The symbolic simulation explores several computational paths; therefore, the simulator must keep track of the state of the temporal assertion for every path.

To solve the above issues, we have developed the procedure described in Fig. 4.

Before starting the simulation, the LTL assertion is translated to a checker in the form of a deterministic finite-state automaton. The automaton always contains a root node as the initial state of the checker and a rejecting node where the assertion fails. The state of a checker is completely identified with an unsigned integer. Each edge is labelled with a propositional formula. Given a checker ch in state s_i and a proposition p_k on the outer edge connecting s_i with s_j; if p_k is true for the current sample, then s_j is the next state of the checker. A sample is a set of couples $S_i = \{(var_1, val_1)_i, ..., (var_n, val_n)_i\}$ where each element $(var_j, val_j)_i$ corresponds to value val_j at time i of variable var_j; $var_1, ..., var_n$ are the variables contained in the LTL assertion. To determine value val_j, the simulator must know the scope in which to find the corresponding variable var_j; therefore, the user has to add such information in the assertion by appending the scope to the variable. In the assertion of Fig. 4, variable a is used as $main :: a$ since it is declared in the main function; likewise, variable in is used without any additional information to specify that it is declared in the global scope. If the simulator tries to make a sample of variable var_k that does not exist in memory at time i, then the sample will contain a val_k equal to 0.

Algorithm 3. Automaton's evaluation

```
1: function evalAutomaton(aut, samp)
2:     for all outEdge in aut.currState.outEdges do
3:         if outEdge.prop.evaluate(samp) then
4:             aut.state = edge.toState
5:             if outEdge.toState.type == Rejecting then
6:                 return false
7:             end if
8:             break
9:         end if
10:    end for
11:    return true
12: end function
```

Function *evalAutomaton* of Algorithm 3 formalises how to perform an evaluation for an automaton *aut* and a sample *samp*. The function searches for an outer edge *outEdge* labelled with a proposition that is true for sample *samp* (line 2–3). After that, the state of the automaton is updated (line 4). If the next state is rejecting (line 5), then the function returns false to notify that the assertion failed (line 6). If the next state is not rejecting, then the function returns true as the assertion did not fail on the current time unit (line 11).

Once the checker and all the utilities to evaluate it on a trace are prepared, we perform symbolic simulation to identify a computational path on which the assertion fails. To do that, we have extended the KLEE framework [27]. In particular, each time a new input must be read in the execution (new symbolic value), the simulator creates a sample of the variables and evaluates the checker on the current time unit. Note that each computational path (called ExecutionState in KLEE) contains a unique instance of the checker stored as an unsigned integer (we only need to keep track of its current state). If the evaluation of $checker_i$ on $path_i$ returns false, then the assertion failed and a faulty execution trace $exec\_trace_i$ is found; otherwise, the simulation continues. As in Sect. 4.1, if the user provided the inputs necessary to make the assertion fail, then only one path is explored by the symbolic simulation.

5.3 Trace Decoration

In this section, we describe how to modify an extracted execution trace to include the information of the failure of a temporal assertion. The result of this procedure is a set of decorated execution traces on which to apply steps 2 and 3 of the methodology described in Sect. 4. To simplify the exposition, we will refer to the example in Fig. 5. The example involves the same implementation reported in listing 1.1 that generates the same execution trace reported in Table 1 on which assertion a_1 fails.

The methodology is based on the assumption that the failure of a temporal assertion can be described as a sequence of propositions $\langle p_1, ..., p_n \rangle$ that are true

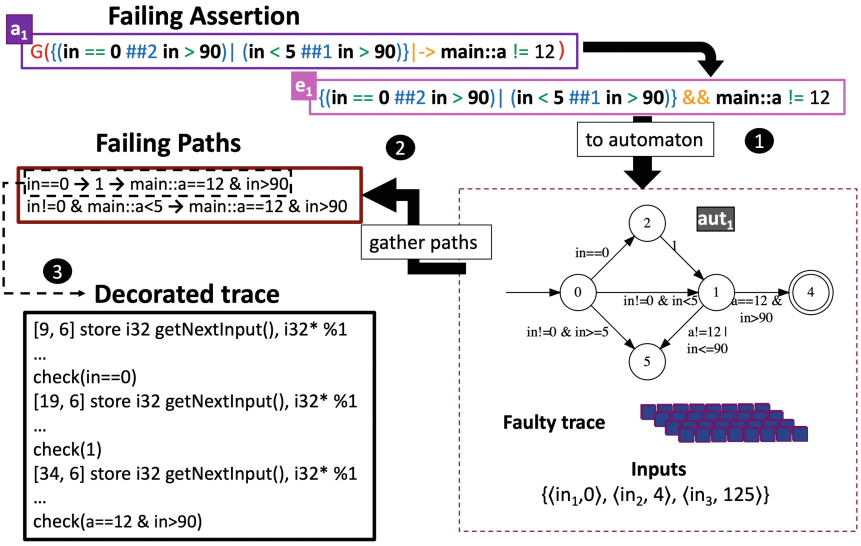

Fig. 5. Trace decoration of the running example

on a sequence of time units $\langle 1, ..., n \rangle$, where p_i is true at time i. For example, assertion a_1 of Fig. 5 fails if the sequence of propositions $\langle in! = 0 \ \& \ a < 5, \ a == 12 \ \& \ in > 90 \rangle$ is true on two consecutive time units. This sequence of propositions corresponds to an accepting path of the automaton generated from the expression $ant \ \& \ !con$, where ant and con is the antecedent and the consequent of the original assertion. The simulator deduces that the assertion fails on the execution trace by checking that all the propositions in the sequence are true on the corresponding time units.

The whole procedure consists of three main steps. First, the original assertion $G(antecedent \rightarrow consequent)$ is converted to the expression $antecedent \ \& \ !consequent$ and translated to an automaton. Note that this automaton contains both accepting and rejecting states. Figure 5 contains the conversion of assertion a_1 to expression e_1 and its translation to automaton aut_1.

In the second step, the procedure retrieves the paths of the automaton justifying the failure of the assertion on the execution trace. This process is formalised in function *retrievePaths* of Algorithm 4. The idea of the algorithm is to evaluate the edges of the automaton using the samples of the execution trace to build the sequences of propositions that make the assertion fail. The inputs of function *retrievePaths* are the automaton *aut* and the list of samples *samps*. Variables *paths* contains the list of retrieved paths, and *currPath* is a utility variable used to build the paths (line 2–3). The algorithm starts by evaluating the edges of the accepting state of the automaton (where the assertion fails) with the last sample of the execution trace (lines 4–6). In the running example, the algorithm starts from state 4 of aut_1 with the sample obtained after the

Algorithm 4. Function to retrieve the paths triggering the failure

```
1: function retrievePaths(aut, samps)
2:     paths = ∅
3:     currPath = ∅
4:     for all inEdge in aut.accState.inEdges do
5:         visitAut(inEdge, aut, samps, paths, currPath, samps.size() − 1)
6:     end for
7:     return paths
8: end function
9:
10: function visitAut(currEdge, aut, samps, paths, currPath, si)
11:     if currEdge.prop.evaluate(samps[si]) then
12:         currPath.push_front(currEdge.prop)
13:         si−−
14:         if currEdge.fromState == aut.rootNode then
15:             paths.push_back(currPath)
16:         else if si >= 0 then
17:             for all inEdge in currEdge.fromNode.inEdges do
18:                 visitAut(inEdge, aut, samps, paths, currPath, si)
19:             end for
20:         end if
21:         si++
22:         currPath.pop_front()
23:     end if
24: end function
```

third input $\langle in3, 125\rangle\}$. For each edge *aut.accState.inEdge*, the algorithm calls function *visitAut*. Among the inputs of *visitAut* we have the edge *currEdge* with which the function is trying to build a path and the index *si* to keep track of which sample must be used to evaluate the proposition on *currEdge*. At line 5, *visitAut* he is called with *si* equal to *sample.size*() − 1 to specify that the path is built from the last sample (last time unit). Function *visitAut* recursively visits the inner edges of each state of *aut* in a DFS fashion (line 10–24). Each time the function manages to build a path that connects the root state with the accepting state of *aut* (line 14), a new path is found and stored in *paths* (line 15). Figure 5 reports the two failing paths retrieved from assertion a_1 in the running example.

In the final step of the procedure, each sequence of propositions is used to generate a decorated execution trace. Formally, a sequence of propositions $\langle p_1, ..., p_n\rangle$ is used to decorate an execution trace with a sequence of checkpoints $\langle c_1, ..., c_n\rangle$ where c_i is a function that returns true if p_i is true at time i, false otherwise. If all checkpoints return true, then the assertion must fail on the execution trace. Figure 5 reports the execution trace decorated with one of the failing paths.

Once a decorated execution trace is generated, we can easily apply the techniques described in the second and third steps of the methodology by considering the differences highlighted below.

- The DPDG must consider the fundamental addresses of all the propositions in the checkpoints
- To determine if an assertion fails on a decorated execution trace, the simulator must verify that all the checkpoints return true.

6 Experimental Results

The proposed methodology has been implemented in an automatic tool extending the KLEE symbolic engine. Its effectiveness and efficency has been evaluated on four well-known C benchmarks compiled to LLVM:

- *xtea* implements the Extended Tiny Encryption Algorithm;
- *matrix mult* is a matrix multiplication algorithm;
- *graph DFS* is a depth first search algorithm;
- *Newton-Raphson* is the famous root finding algorithm.

The experimental results have been carried out on a 2.9 GHz Intel Core i7 processor equipped with 16 GB of RAM and running Ubuntu 20.04 LTS.

Table 2 reports the results in terms of execution time and reduction quality referred to an execution trace exposing a bug for each design. In particular, Table 2 compares the results of our tool with a *baseline* obtained by applying the best achievable reduction, that is, by manually inspecting the trace and removing the unnecessary instructions; indeed, this procedure can be performed only on short traces. The second column (*Original length*) reports the length of the original execution trace that makes the assertion fail, before applying any reduction. The third column (*Our approach*) reports the final length of the trace after applying our approach. The fourth column (*Manual Inspection*) reports the baseline. The fifth column reports the reduction quality as a ratio between "Manual inspection" and "Our approach". Here we can observe that our tool produces results very close to the baseline (reduction quality close to 1) for all the reported tests. The last column reports the execution time of our tool.

Table 3, instead, shows the scalability of our approach. It reports, for the *Netwon-Raphson* benchmark, the reduction percentage and the execution time at the increasing of the length of the target execution trace. These results show that our tool is capable, in a few seconds, of providing a reduction of over 60% of the original trace, even for traces hundreds of instructions long.

Table 2. Analysis of the reduction quality

Name	Original length	Reduced length		Reduction quality	Reduction time
		Our approach	Manual inspection		
xtea	190	155	155	1	1830 ms
Matrix mult	150	127	122	0.96	2631 ms
Newton-Raphson	213	76	76	1	2056 ms
Graph DFS	236	207	205	0.99	4623 ms

Table 3. Analysis of the approach's scalability

Original length	Reduced length	Reduction time	Reduction
482	154	3 s	68.05%
1379	389	36 s	71.79%
10283	2888	437 s	71.91%

7 Conclusions

In this paper, we presented a new methodology and a related tool to automatically remove irrelevant instructions from execution traces identifying unexpected behaviours in system-level designs. Starting from an unexpected behaviour formalised through an assertion, the tool generates a reduced execution trace that triggers such behaviour, thus highlighting the essential instructions related to it. To achieve that, we perform a preliminary reduction involving a DPDG and dynamic program slicing; then, the remaining instructions are further reduced through an instruction clusterization procedure. One of the main aspects of our methodology is that we verify by simulation if the remaining trace is still capable of triggering the unexpected behaviour; therefore, the output trace corresponds to an executable program.

After that, the methodology was modified to support temporal behaviours. This last extension opens a whole new world of possibilities, allowing the application of old and new program slicing techniques to systems implementing functional behaviours described by means of LTL formulas.

Experimental results show the effectiveness and scalability of the approach.

References

1. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceeding of CGO (2004)
2. Bragaglio, M., Donatelli, N., Germiniani, S., Pravadelli, G.: System-level bug explanation through program slicing and instruction clusterization. In: 2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC), pp. 1–6 (2021)

3. Weiser, M.: Program slicing. In: IEEE Transactions on Software Engineering, vol. SE-10, no. 4, pp. 352–357 (1984)
4. Ottenstein, K. J., Ottenstein, L.M.: The program dependence graph in a software development environment. In: Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ser. SDE 1. New York, NY, USA. Association for Computing Machinery, pp. 177–184 (1984). https://doi.org/10.1145/800020.808263
5. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987)
6. Bergeretti, J.-F., Carré, B.A.: Information-flow and data-flow analysis of while-programs. ACM Trans. Program. Lang. Syst. **7**(1), 37–61 (1985). https://doi.org/10.1145/2363.2366
7. Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure, B., Wolfe, M.: Dependence graphs and compiler optimizations. In: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL 1981, New York, NY, USA. Association for Computing Machinery, pp. 207–218 (1981). https://doi.org/10.1145/567532.567555
8. Reps, T., Bricker, T.: Illustrating interference in interfering versions of programs. In: Proceedings of the 2nd International Workshop on Software Configuration Management, ser. SCM 1989, New York, NY, USA. Association for Computing Machinery, pp. 46–55 (1989). https://doi.org/10.1145/72910.73347
9. Korel, B., Laski, J.: Dynamic slicing of computer programs. J. Syst. Softw. **13**(3), 187–195 (1990)
10. Chen, T., Cheung, Y.: Dynamic program dicing. In: Proceeding of IEEE CSM, pp. 378–385 (1993)
11. Renieres, M., Reiss, S.: Fault localization with nearest neighbor queries. In: Proceeding of IEEE ASE, pp. 30–39 (2003)
12. Wong, W.E., Qi, Y.: Effective program debugging based on execution slices and inter-block data dependency. J. Syst. Softw. **79**(7), 891–903 (2006)
13. Germiniani, S., Danese, A., Pravadelli, G.: Automatic generation of assertions for detection of firmware vulnerabilities through alignment of symbolic sequences. In: IEEE Transactions on Emerging Topics in Computing (2020)
14. Duanzhi, C.: A collection of program slicing. In: Proceeding of ICCASM (2010)
15. Field, J., Ramalingam, G., Tip, F.: Parametric program slicing. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL 1995, New York, NY, USA. Association for Computing Machinery, pp. 379–392 (1995). https://doi.org/10.1145/199448.199534
16. Field, J., Tip, F.: Dynamic dependence in term rewriting systems and its application to program slicing. In: Hermenegildo, M., Penjam, J. (eds.) PLILP 1994. LNCS, vol. 844, pp. 415–431. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58402-1_29
17. J. Q. Ning, J.Q., Engberts, A., Kozaczynski, W.: Automated support for legacy code understanding. Commun. ACM, **37**(5), 50–57 (1994). https://doi.org/10.1145/175290.175295
18. Venkatesh, G.A.: The semantic approach to program slicing. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, ser. PLDI 1991, New York, NY, USA. Association for Computing Machinery, pp. 107–119 (1991). https://doi.org/10.1145/113445.113455
19. Kramkar, M., Fritzson, P., Shahmehri, N.: Interprocedural dynamic slicing applied to interprocedural data flow testing. In: Conference on Software Maintenance, vol. 1993, pp. 386–395 (1993)

20. Choi, J.-D., Miller, B.P., Netzer, R.H.B.: Techniques for debugging parallel programs with flowback analysis. ACM Trans. Program. Lang. Syst. **13**(4), 491–530 (1991). https://doi.org/10.1145/115372.115324
21. Liu, C., Fei, L., Yan, X., Han, J., Midkiff, S.: Statistical debugging: a hypothesis testing-based approach. IEEE Trans. Softw. Eng. **32**(10), 831–848 (2006)
22. Liblit, B., Naik, M., Zheng, A. X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. In: Proceeding of ACM SIGPLAN PLDI, pp. 15–26 (2005)
23. Wen, W.: Software fault localization based on program slicing spectrum. In: Proceeding of IEEE ICSE, pp. 1511–1514 (2012)
24. Wang, T., Roychoudhury, A.: Hierarchical dynamic slicing. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis, ser. ISSTA 2007, New York, NY, USA. Association for Computing Machinery, pp. 228–238 (2007). https://doi.org/10.1145/1273463.1273494
25. https://llvm.org/docs/LangRef.html
26. Standard for property specification language (PSL). In: IEC 62531:2012(E) (IEEE Std 1850–2010), pp. 1–184 (2012)
27. Cadar, C., Dunbar, D., Engler, D.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceeding of USENIX OSDI (2008)
28. Harrold, M.J., Malloy, B., Rothermel, G.: Efficient construction of program dependence graphs. In: Proceeding of ACM ISSTA, pp. 160–170 (1993)