Using Relational Verification for Program Slicing

Bernhard Beckert[1(✉)], Thorsten Bormer[1], Stephan Gocht[2], Mihai Herda[1(✉)], Daniel Lentzsch[1], and Mattias Ulbrich[1(✉)]

[1] Karlsruhe Institute of Technology, Karlsruhe, Germany
{beckert,herda,ulbrich}@kit.edu
[2] KTH Royal Institute of Technology, Stockholm, Sweden

Abstract. Program slicing is the process of removing statements from a program such that defined aspects of its behavior are retained. For producing precise slices, i.e., slices that are minimal in size, the program's semantics must be considered. Existing approaches that go beyond a syntactical analysis and do take the semantics into account are not fully automatic and require auxiliary specifications from the user. In this paper, we adapt relational verification to check whether a slice candidate obtained by removing some instructions from a program is indeed a valid slice. Based on this, we propose a framework for precise and automatic program slicing. As part of this framework, we present three strategies for the generation of slice candidates, and we show how dynamic slicing approaches – that interweave generating and checking slice candidates – can be used for this purpose. The framework can easily be extended with other strategies for generating slice candidates. We discuss the strengths and weaknesses of slicing approaches that use our framework.

Keywords: Program slicing · Relational verification

1 Introduction

Program slicing, introduced by Weiser [40], is a technique to reduce the size of a program while preserving a certain part of its behavior. Different kinds of slicing approaches have been developed [31]. A *static slice* preserves the program's behavior for all inputs, while a *dynamic slice* preserves it only for a particular single input. A *backward slice* keeps only those parts of the program that influence the value of certain variables at a certain location in the program, while a *forward slice* keeps those program parts whose behavior is influenced by the variables' values. The form of slicing introduced by Weiser is now known as *static backward slicing* and is the form of slicing which is pursued in this paper. Slicing techniques can be used to optimize the results of compilers. Slicing is also a powerful tool for challenges in software engineering such as code comprehension, debugging, refactoring, and fault localization [8], as well as in information-flow security [19].

P. C. Ölveczky and G. Salaün (Eds.): SEFM 2019, LNCS 11724, pp. 353–372, 2019.
https://doi.org/10.1007/978-3-030-30446-1_19

```
1  int f(int h, int N){
2    int i = 0;
3    int x = 0;
4    while(i < N) {
5      if(i < N - 1)
6        x = h;
7      else
8        x = 42;
9      i++;
10   }
11   return x;
12 }
```

```
1  int f(int h, int N){
2    int i = 0;
3    int x = 0;
4    while(i < N) {
5      if(i < N - 1)
6        skip;
7      else
8        x = 42;
9      i++;
10   }
11   return x;
12 }
```

```
1  int f(int h, int N){
2    int i = 0;
3    int x = 0;
4    while(i < N) {
5      if(i < N - 1)
6        skip;
7      else
8        skip;
9      i++;
10   }
11   return x;
12 }
```

Fig. 1. (a) Original program, (b) slice w.r.t. variable x at line 11, (c) incorrect slice candidate

All applications of slicing can benefit from small and precise slices. Most existing slicing approaches, however, are only syntactical, i.e., they do not take the semantics of the various program operations into account. On the other hand, many existing approaches that do take the semantics into account are not fully automatic and require auxiliary specifications from the user (e.g., precomputed or user-provided functional loop invariants are used in [4,22]).

Figure 1 shows an example of static backward slicing. The goal is to slice the program in Fig. 1a w.r.t. a slicing criterion which requires the value of x at the statement in line 11 to be preserved. A valid slice for this criterion is shown in Fig. 1b: The assignment in line 6 of the program has been removed. This line has no effect on the value of x, as it is always set to 42 in the last loop iteration. In fact, the statement is not completely removed but replaced with an effect free *skip* statement to keep the program's structure similar to that of the input program. To show that this program is a valid slice, a syntactical analysis is insufficient, as it would not be able to see that in the *last* iteration variable x is overwritten. A semantic analysis is required to determine that the last loop iteration always executes the else-branch. The slicing procedure needs to reason about loops and path conditions, and in this paper we use relational verification for this purpose.

Relational verification approaches that consider the program's semantics and automatically reason about loops have become available in the last couple of years, e.g. [13,24,38]. These approaches can efficiently and automatically show the equivalence of two programs – provided that the two programs have a similar structure. Since slices are constructed by removing program statements, they have a similar structure to the original program and are a good use case for relational verification. In this paper we make the following *contributions*:

1. We provide an extensible framework for precise and automatic slicing of programs written in a low level *intermediate representation* language, as well as a semantics therefor. The slicing approaches using this framework need no (auxiliary) specification other than the slicing criterion.
2. We adapt a relational verifier to check if a slice candidate obtained by removing instructions from a program is a valid slice.
3. We adapt a dynamic slicing algorithm and use it to generate slice candidates.

The feasibility of our framework has been shown in a tool paper [5] describing an implementation. Here, we focus on the theoretical background of the framework.

Structure of the Paper. In Sect. 2, we formally describe the programs which we handle and define what a valid slice is. We introduce relational verification in Sect. 3 and extend it to prove the validity of a slice candidate. The framework itself, as well as three slicing approaches based on this framework are described in Sect. 4. Section 5 consists of a discussion of the framework. We present related work in Sect. 6 and conclude in Sect. 7.

2 Static Backward Slicing

Static backward slicing as introduced by Weiser [40] reduces a program by removing instructions in a way that preserves a specified subset of the program's behavior. The *slicing criterion* – the specification of the behavioral aspects that must be retained – is given in form of a set of program variables and a location within the program. Instructions may be removed if and only if they have no effect (a) on the value of the specified program variables at the specified location whenever it is reached and (b) on how often the location is reached.

High level programming languages are feature rich, increasing the effort needed for a program analysis. A solution for dealing with language complexity is to perform the analysis on a simpler, intermediate representation. While the implementation of our slicing framework [5] works on LLVM IR [1] programs, to keep the definitions in this paper easy to understand, we here use a language whose computational model is similar to that of LLVM IR but that has only four instructions: *skip*, *halt*, *assign*, and *jnz*. We formalize the notions of *slice candidate*, *slicing criterion* and *valid slice* using a computation model based on a register machine with an unbounded number of registers. Thus we do not have high-level constructs such as *if* or *while* statements but instead branching and looping are done using conditional jump instructions. The advantage of using such a language is the fact that the control flow is reduced to jumps, and, in the context of slicing, a program remains executable no matter what statements are removed. Figure 2 shows the examples from Fig. 1 written in our simple IR language. The criterion location is now 12, the criterion variable is still x.

```
0  assign i 0
1  assign x 0
2  assign c1 (i >= N)
3  jnz c1 12
4  assign t1 (N - 1)
5  assign c2 (i >= t1)
6  jnz c2 9
7  assign x h
8  jnz 1 10
9  assign x 42
10 assign i (i + 1)
11 jnz 1 2
12 halt
```

```
0  assign i 0
1  assign x 0
2  assign c1 (i >= N)
3  jnz c1 12
4  assign t1 (N - 1)
5  assign c2 (i >= t1)
6  jnz c2 9
7  skip
8  jnz 1 10
9  assign x 42
10 assign i (i + 1)
11 jnz 1 2
12 halt
```

```
0  assign i 0
1  assign x 0
2  assign c1 (i >= N)
3  jnz c1 12
4  assign t1 (N - 1)
5  assign c2 (i >= t1)
6  jnz c2 9
7  skip
8  jnz 1 10
9  skip
10 assign i (i + 1)
11 jnz 1 2
12 halt
```

Fig. 2. The three examples from Fig. 1 translated into our IR language.

$$\frac{P[pc] = skip}{(s, pc) \rightsquigarrow (s, pc + 1)} \qquad \frac{P[pc] = halt}{(s, pc) \rightsquigarrow (end, pc)}$$

$$\frac{pc > len(P)}{(s, pc) \rightsquigarrow (end, pc)} \qquad \frac{}{(end, pc) \rightsquigarrow (end, pc)}$$

$$\frac{P[pc] = jnz \;\; v \;\; target \qquad s(v) = 0}{(s, pc) \rightsquigarrow (s, pc + 1)} \qquad \frac{P[pc] = assign \;\; v \;\; exp \qquad x = s(exp)}{(s, pc) \rightsquigarrow (s[v \backslash x], pc + 1)}$$

$$\frac{P[pc] = jnz \;\; v \;\; target \qquad s(v) \neq 0}{(s, pc) \rightsquigarrow (s, target)}$$

Fig. 3. The semantics of our programming language for a fixed program P

We will now define the semantics of our IR language. Let *Var* be the set of program variables, S the set of states, where a state is a function $s : Var \to \mathbb{N}$, and $pc \in \mathbb{N}$ the program counter. An instruction I is an atomic operation that can be executed by the machine. Let $\mathcal{I}$ be the set of all four instructions provided by our IR language. When an instruction is executed, the system changes its state and program counter as determined by the transition function $\rho : S \times \mathbb{N} \times \mathcal{I} \to S \times \mathbb{N}$. A program P is a finite sequence of instructions: $\langle I_0, I_1, \ldots I_n \rangle$. We denote a location i of program P as $P[i]$ with $P[i] = I_i$ for any $i \in \{0, 1, \ldots n\}$ with $0 \leq i \leq len(P) - 1$, where $len(P)$ is the length of the program.

The semantics of the four instructions in our IR language is shown in Fig. 3. The instruction *skip* increments the program counter and has no other effects. To obtain a slice candidate, instructions in the original program are replaced with *skip*. To model the termination of programs we introduce a special state, *end*, such that once the system reaches this state, it will remain in this state forever. The instruction *halt* is used to bring the system to the *end* state. The assignment instruction, *assign*, takes a variable v and an integer expression *exp* as arguments. After the execution of this instruction, the value of the variable v in the new state is updated with the result x of the expression *exp* and the program counter is incremented. To obtain precise slices, we restrict *exp* to only one operator. The conditional jump instruction, *jnz*, allows the register machine to support branching and looping. The instruction gets a variable v and an integer expression *target* as arguments. If the variable v evaluates to zero in the state in which *jnz* is executed, then the program counter is incremented, otherwise the program counter is set to the value of *target*. We will now define program traces:

Definition 1 (Program trace). *A trace T of a program P is a possibly infinite sequence of state and program counter pairs $\langle (s_0, pc_0), (s_1, pc_1), \ldots \rangle$ such that:*

1. $pc_0 = 0$
2. *For each trace index i but the last, $(s_i, pc_i) \rightsquigarrow (s_{i+1}, pc_{i+1})$*

We use $T^s[i]$ and $T^{pc}[i]$ to denote respectively the ith state and the ith program counter of a trace. Also we use $len(T) \in \mathbb{N} \cup \{\omega\}$ to denote the length of trace $\mathcal{T}$; note that it can be infinite. We define F_T^l to be the sequence comprised of those states $T^s[i]$ for which $T^{pc}[i] = l$, in the same order as they appear in T^s. We define the notions of a slicing criterion, slice candidate and valid slice:

Definition 2 (Slicing Criterion). *A slicing criterion C for a program P is a pair (i_C, Var_C) where i_C is a location in P and $Var_C \subseteq Var$.*

Definition 3 (Slice Candidate). *A slice candidate for a program P_o is a program P_L that is constructed by replacing the instructions at some locations in P_o with the skip instruction. That is, given a set L of locations of program P_o:*

$$P_L[i] = \begin{cases} skip, & i \in L \\ P_o[i], & i \notin L \end{cases}$$

Definition 4 (Valid Slice). *Given a slicing criterion (i_C, Var_C), a slice candidate P_s for a program P_o is a valid slice for P_o if, for any two traces T_s of P_s and T_o of P_o with $T_s[0] = T_o[0]$, the following holds:*

1. $len(F_{T_o}^{i_C}) = len(F_{T_s}^{i_C})$,
2. $F_{T_o}^{i_C}[i](v) = F_{T_s}^{i_C}[i](v)$ *for every* $v \in Var_C$ *and every* i *with* $0 \leq i < len(F_{T_o}^{i_C})$.

The first requirement ensures that the criterion location is reached in both the original program and the slice candidate the same number of times. The second requirement ensures that the criterion variables have the same values every time the criterion location is reached in the original program and in the slice candidate.

Weiser [40] deals with the feature-richness of programming languages by working on flow graphs, and slices are constructed by removing nodes from the flow-graph. In his approach, however, only nodes with a single successor can be removed while we can remove conditional jumps. Definition 4 is similar to the concept of observation windows in [40]; however, we do not require the original program to terminate. Thus, we extend the definition of Weiser to nonterminating programs, as opposed to many other slicing approaches (as stated in [34]) that are not termination sensitive. Compared to other extensions of the definition of Weiser, e.g. the one in [3], Definition 4 allows for slices which are not *quotients* of the original program, i.e., it allows the removal of conditional jumps while preserving the instructions which are in the program locations between the conditional jump and the jump target. The program

```
0  assign x 42
1  halt
```

is thus a valid slice of the program shown in Fig. 2a, according to Definition 4. Not requiring the slice to be a quotient allows the removal of additional statements. However, the structure of a slice may differ significantly from that of the

original program. When using slicing with the goal of program optimization a further reduction of the program is a clear advantage. If the goal is program comprehension, however, then the slice not being a quotient of the original program presents both advantages and disadvantages. One the one hand, a significantly different structure of the slice compared to that of the original program, may cause the user to have difficulties understanding the behavior of the original program. On the other hand, the fact that some conditional jump statements are not in the slice may indicate to the user that certain program branches are irrelevant with respect to the given slicing criterion and help him better understand the program behavior.

3 Relational Verification of Slice Candidates

Relational verification is an approach for establishing a formal proof that if a relational precondition holds on two respective pre-states of two programs P and Q then the respective post-states of P and Q will fulfill a relational postcondition. For two complex programs that yet are similar to each other, much less effort is required to prove their equivalence than to prove that they both satisfy a complex functional specification. The effort for proving equivalence mainly depends on the difference between the programs and not on their overall size and complexity. This is particularly beneficial for the verification of slice candidates, because the candidates are obtained by replacing program instructions with *skip* and thus have a structure similar to the original program.

We formally define the property that is checked by a relational verifier. To that end, we call a predicate π a transition predicate for a program P if for any two states, s and s', $\pi(s, s')$ holds if and only if program P when started in state s terminates in state s'. Thus, for two programs, P and Q, a relational verifier checks the validity of the following proof obligation:

$$Pre(s_P, s_Q) \wedge \pi(s_P, s'_P) \wedge \rho(s_Q, s'_Q) \rightarrow Post(s'_P, s'_Q),$$

where π and ρ are transition predicates for P and Q, respectively, and *Pre* and *Post* are respectively the relational precondition and postcondition.

However, a relational verifier that only checks this property is of limited use for checking slice candidates. For the case in which the location of the slicing criterion refers to the post-state (in Fig. 2a that corresponds to location 12 that contains the *halt* instruction), relational verification can be used to check whether a slice candidate is a valid slice. For a slice candidate Q obtained from a program P, this is done by setting *Pre* to require equal pre-states s_P and s_Q and *Post* to require the criterion variables to evaluate to the same values in the post-states s'_P and s'_Q. However, a successful proof shows the validity of the slice candidates only for inputs for which both P and Q terminate, as the transition predicates may be false for certain pre-states. In the rest of this section we show how a relational verifier can be adapted to support slicing on locations other than the end of the program and how to use relational verification to also show that

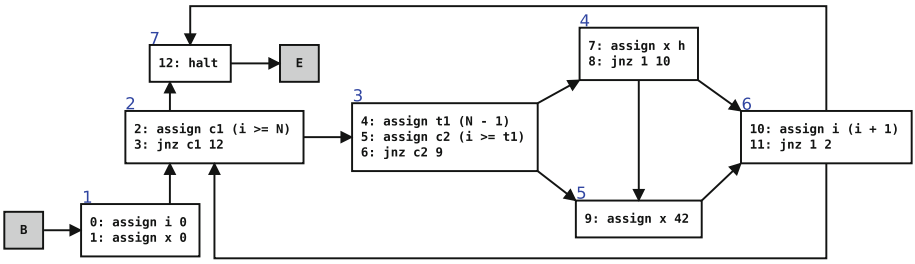

Fig. 4. The CFG for the program in Fig. 2a

the program and candidate run in lockstep (i.e. the two executions run through corresponding states), ensuring thus mutual termination.

Our slicing framework is based on the LLRêve [14,24] relational verifier, which works on programs written in LLVM IR. It analyzes the control flow graphs (CFGs) of the programs and reduces the validity of the relational specification to the satisfiability of a set M of Horn-constraints over uninterpreted predicates. The satisfiability of the Horn-constraints in M can be checked with state of the art SMT solvers such as Z3 [32] and Eldarica [35].

If the analyzed programs contain loops, their CFGs contain cycles, which constitute a challenge for verification because the number of iterations is unknown. LLRêve handles cycles by using so called *synchronization points*, at which the program state is abstracted by means of predicates. The paths between synchronization points are cycle free and can be handled easily. Synchronization points are defined by labeling basic blocks of the CFG with unique numbers. The entry and the exit of a function are considered special synchronization points B and, respectively, E. Additionally, the user can also define synchronization points at any location of the analyzed programs. The user must ensure that there is a synchronization point for each basic block of the CFG of the two programs, and has to match them appropriately. In general, it is difficult to find matching synchronization points for two programs; however, in the case of program slicing this can be done automatically by keeping the CFG of the original program. Figure 4 shows the CFG for the program in Fig. 2a and each basic block is labeled with the number of a synchronization point. In the CFG of the slice in Fig. 2b, the *assign* instruction in block 4 is replaced with *skip*, the synchronization points remain the same, and matching them is trivial. If a conditional jump is replaced with *skip*, we only remove the edge to the block containing the jump target, thus keeping the same synchronization points for the slice candidate.

Given one synchronization point per basic block, the CFG can be viewed as a set of linear paths $\langle n, \pi, m \rangle$, where n and m denote the starting and end synchronization points of the path, and $\pi(s, s')$ is the transition predicate between the two synchronization points, with s and s' being the states before and, respectively, after the transition. Because the linear paths consists of assignments only, the transition predicates can be easily computed. For two programs with a similar structure, it is expected that there exist coupling predicates that describe

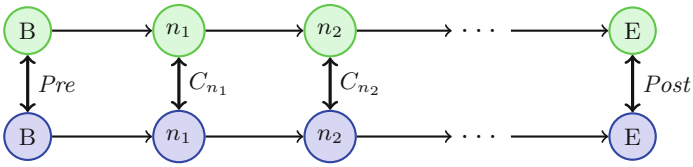

Fig. 5. Illustration of coupled control flow of two fully synchronized programs

the relation between the program states at two corresponding synchronization points. For two programs P and Q we introduce an uninterpreted coupling predicate $C_n(s_p, s_q)$ for each synchronization point n, as shown in Fig. 5. The relational precondition *Pre* and postcondition *Post* are the coupling predicates for the special synchronization points B and E, respectively. The set M consists of Horn-constraints over these coupling predicates. For two linear paths between synchronization points n and m in programs P and Q characterized by the two transition predicates π and ρ, respectively, this constraint is added to M:

$$C_n(s_p, s_q) \wedge \pi(s_p, s'_p) \wedge \rho(s_q, s'_q) \rightarrow C_m(s'_p, s'_q) \tag{1}$$

To ensure that there is no divergence from lockstep, for every two paths $\langle n, \pi, m\rangle$ and $\langle n, \rho, k\rangle$ in programs P and Q, respectively, with $m \neq k, m \neq n, n \neq k$ the following constraint is added to M:

$$C_n(s_p, s_q) \wedge \pi(s_p, s'_p) \wedge \rho(s_q, s'_q) \rightarrow \mathit{false} \tag{2}$$

Note, that even though the synchronization points m and k do not appear in Eq. 2, they respectively determine the transition predicates π and ρ.

Theorem 1. *Let P and Q be programs specified with the relational precondition Pre and postcondition Post, for which matching synchronization points have been found. Let M be the set of constraints generated according to 1 and 2. If M is satisfiable, then for every pair of pre-states satisfying Pre:*

1. *The synchronization points are reached in the same order in P and Q,*
2. *If P terminates, then so does Q and Post holds for the two post-states.*

Proof. For distinct synchronization points n, m, k, the fact that constraint 2 has a model implies that *(case 1)* π or ρ is *false*, meaning that the execution of P or Q cannot reach respectively m or k from n, or *(case 2)* C_n is false meaning that n is not reachable in P or Q, or per (chaining of) constraint 1 the pre-states do not satisfy the precondition. Thus, P and Q reach the synchronization points (including E, thus implying mutual termination) in the same order. For two synchronization points n, m, the fact that constraint 1 has a model implies that *(case 1)* m cannot be reached from n in P or Q, or *(case 2)* C_n is false and n is not reachable or the pre-states do not satisfy the precondition, or *(case 3)* starting in n with C_n holding, both programs reach m and C_m holds there. The constraints generated according to 1 are thus interpolants that show the validity of the relational specification. □

To check the validity of a slice candidate for the cases in which the criterion location is in the middle of the program, we adapt the constraints generated by the relational specification. The relational precondition *Pre* still requires equal pre-states, while the relational postcondition *Post* is set to *true*. We ensure a synchronization point n_C exists in the program and slice candidate at the location of the criterion instruction. For example Fig. 2a n_C is the synchronization point 5 in Fig. 4. If the criterion location is part of a basic block with more than one instruction, we split that basic block up such that we obtain a block containing only the criterion location. For a program P with a slice candidate Q and a given slicing criterion (i_C, V_C) with a synchronization point n_C we add the following constraint:

$$C_{n_C}(s_P, s_Q) \rightarrow \forall x \in V_C \;\; s_P(x) = s_Q(x) \tag{3}$$

Theorem 2. *Let P be a program and Q a slice candidate specified with the relational precondition Pre requires equal pre-states and postcondition Post is true. Let M be the set of constraints generated according to 1, 2 and 3. If M is satisfiable, then for every pair of pre-states that fulfill Pre:*

1. *The criterion location is reached equally often in P and Q,*
2. *At the i-th time (for $i \geq 1$) the criterion instruction is reached in P and in Q, the criterion variables are equal in P and Q,*
3. *If P terminates, then so does Q.*

Proof. From Theorem 1 results that P and Q run in lockstep with respect to the synchronization points. The instruction at the criterion location has its own synchronization point. As a consequence of this, the criterion instruction is executed in both P and Q the same number of times and the candidate terminates iff the original program terminates. Due to Constraint 3, the coupling predicate corresponding to the criterion locations ensures that each time the criterion location is reached, the criterion variables have the same values. □

Thus, for a program P with a slice candidate Q and a slicing criterion (i_C, V_C), if the set M containing the constraints 1, 2 and 3 for every synchronization point is satisfiable, then Q is a valid slice according to Definition 4. Moreover, if the set M is unsatisfiable, then the SMT solver returns an unsatisfiability proof that contains a counterexample with two concrete inputs for which the slice property is violated – provided the SMT solver does not time out.

4 A Framework for Automatic Slicing

Being able to use relational verification to check whether a slice candidate is valid, we construct a framework for automatic program slicing. The framework consists of two components which interact with each other. The first component, the candidate generation engine, generates the slice candidates and sends them to the second component, the relational verifier (in this case LLRêve).

The relational verifier transmits one of three possible answers to the candidate generation engine: (1) the candidate is a valid slice, (2) the candidate is not valid along with an input that leads to a violation of the slice property (Definition 4), or (3) a timeout. The candidate generation engine can use the answer to adapt its candidate generation strategy.

An advantage of the framework is that the candidate generation engine does not need to care about the correctness of the slice candidates it generates – as this is taken care of by the relational verifier. The framework can easily be extended with candidate generation strategies other than those that we present in this paper. Thus, it provides a platform for relational verification based slicing for the software slicing community.

We distinguish between two types of candidate generation strategies. On the one hand there are strategies that generate candidates by replacing program instructions by *skip* according to some heuristics without using any information from the relational verifier other than the existence of a counterexample. Examples for such properties are described in Sect. 4.1. On the other hand there are strategies that also consider the values from the counterexample when generating the next slice candidates. We present one such strategy, *counterexample guided slicing*, in Sect. 4.2.

4.1 Removing Instructions Based on Heuristics

The *brute forcing* (BF) strategy generates all possible slice candidates. As their number is exponential w.r.t. the number of instructions in the original program, it is clear that this strategy does not scale for large programs. Nevertheless, this strategy has the benefit of generating the smallest possible slice with our framework. Brute forcing can be used as part of a divide and conquer strategy to slice parts of programs which are small enough. As an improvement, this strategy can start by generating the candidates in ascending order with respect to their size, i.e. the number of instructions that the candidate retains from the original program. Once a candidate is shown to be a valid slice, no further candidates need to be checked, as their size cannot be smaller than that of the found slice.

The *single statement elimination* (SSE) strategy successively replaces a single instruction of the original program with `skip`, and checks whether the obtained program is a valid slice. If this is the case, the strategy attempts to successively remove every other instruction as well. The strategy requires, in the worst case, quadratically many calls to the relational verifier, which occurs when in each iteration the last candidate is shown to be a valid slice. Although this approach scales better than BF, it finds only slices in which program instructions can be removed individually. Groups of instructions such as `assign x (x + 50)` and `assign x (x - 50)` where the removal of a single instruction results in an invalid slice candidate, but removing the entire group would result in a valid slice cannot be removed. The SSE strategy can be generalized to support the removal of groups of up to a given number of instructions.

4.2 Counterexample Guided Slicing

The *counterexample guided slicing* (CGS) strategy uses *dynamic slicing* to generate slice candidates. Dynamic slicing was first introduced in [27], and a survey on dynamic slicing approaches can be found in [28]. For the CGS strategy we adapted the dynamic slicing algorithm from [2], which is a syntactic approach based on the Program Dependence Graph (PDG) [15]. The PDG is a directed graph in which nodes represent program instructions, conditions, or input parameters, and edges represent possible dependencies between the nodes. An edge from node n_1 to node n_2 encodes that n_1 may depend on n_2. There are roughly two types of dependencies in the PDG. On one hand data dependencies arise when one node uses program variables which are defined in another node. Control dependencies, on the other hand, arise when the execution of a node depends on the other, control, node (e.g. an instruction may be executed only if the condition of a conditional jump is *true*). Whether an edge exists between two nodes in the PDG is determined syntactically by analyzing the CFG. Because the CFG represents an over-approximation of the possible program executions, the PDG edges also represent an over-approximation of the real dependencies in the program. Using the PDG, a backward slice is computed by finding all nodes that are reachable from a node representing the criterion location. On the most basic level, the algorithm in [2], which receives the PDG and an execution trace as inputs, works by computing the subgraph of the PDG which contains only the nodes corresponding to those instructions which have been executed in the program trace. The dynamic slice is computed using this subgraph and further optimizations are possible, as it has to be valid only for a single input.

A PDG node can depend on multiple other nodes, but some of these dependencies are determined by the execution path of the program (e.g. a variable can be assigned on more than one branch, resulting in multiple dependencies for instructions that use that variable). Unlike static slicing, for dynamic slicing only one execution path is relevant – the one corresponding to the input for which the dynamic slice is computed. Thus, PDG edges representing dependencies that are relevant only for other inputs can be removed. A similar situation arises with loops: at different loop iterations, a node inside the loop body may have different dependencies. When performing dynamic slicing, the number of iterations done by a loop is known (assuming the program terminates for the input), and the PDG can be extended with nodes representing the body instructions at different iterations, which also leads to an increased precision of the dynamic slice. The extended PDG is called a *dynamic dependence graph* (DDG) in [2]. Based on the observation that the nodes inside the loop body can depend on only a finite number of other nodes, a new node is added to the PDG just for those iterations in which the corresponding instruction has different dependencies than in all previous iterations. These optimizations give rise to the *reduced dynamic dependence graph* (RDDG). Thus, by ignoring dependencies caused by other inputs than the one for which the dynamic slice is computed, additional instructions can be removed than in the case of static slicing. To ensure compatibility with the slicing property from Definition 4, we adapt this algorithm to support cri-

Data: Program P, Slicing criterion (i_C, V_C)
Result: Program Slice P_s
$P_s \leftarrow \Phi$; $s \leftarrow \bar{0}$; $b \leftarrow \mathit{false}$;
repeat
  $P_d \leftarrow \mathit{dynamicSlice}(P, s, (i_C, V_C))$;
  $P_s \leftarrow \mathit{SDS}(P_s, P_d)$;
  $(b, s) \leftarrow \mathit{relationalVerification}(P, P_s, (i_C, V_C))$;
until $b \vee \mathit{timeout}$;
if *timeout* **then**
  $P_s \leftarrow P$;
end
return P_s;

Algorithm 1. The CGS Strategy

terion locations other than the end of the program. For this, when computing the dynamic slice with the RDDG we do not mark the `return` statement, as is done in [2], but rather all nodes that correspond to the criterion location. If the criterion location is inside a loop, then multiple nodes are marked.

The adapted RDDG dynamic slicing algorithm is purely syntactical and thus scales much better than a semantic approach. Thus we can use it as part of the candidate generation strategy, as relational verification of slice candidates remains the bottleneck of our framework.

For the CGS strategy we wish to merge several dynamic slices $P_{d_1}, \ldots, P_{d_n}$ for the respective input states $s_1, \ldots, s_n$ into a single dynamic slice P_u that is a valid for all inputs $s_1, \ldots, s_n$. In general, the union slice of dynamic slices (which contains all program instructions that are in at least one dynamic slice) is not a correct dynamic slice for all respective inputs of the given dynamic slices. A solution to this was presented in [18] in the form of an iterative algorithm called *simultaneous dynamic slicing (SDS)*, which computes a single dynamic slice valid for each input in a given set.

We can now present the CGS strategy, shown in Algorithm 1. It starts with an initialization of the slice candidate P_s with a program Φ, in which all instructions have been replaced with *skip*, of an arbitrary initial state s, e.g. one in which all variables are set to 0 and of the variable b which will be set to true when a valid slice will be found. The strategy uses the initial state s with the criterion (i_C, V_C) to compute a dynamic slice P_d. The instructions from P_d are then added to the slice candidate P_s which is checked for validity by the relational verifier. If P_s is a valid slice candidate, the variable b is set to *true* and the strategy returns P_s. Otherwise, the relational verifier delivers a counterexample, which is used as the initial state s in the next iteration. Both the dynamic slicer and relational verifier may timeout, in which case the strategy returns the original program P.

Theorem 3. *Let P be a program and P_d be a dynamic slice for all initial states $s \in S_d$, and s_{ce} be the counterexample obtained when checking whether P_d is a valid slice of P. Then the following holds:*

1. $s_{ce} \notin S_d$.
2. *The dynamic slice* P_{ce} *for the initial state* s_{ce} *contains at least one instruction which is not in* P_d.

Proof. Both properties follow from the correctness of the relational verifier and of the dynamic slicer and of the SDS algorithm. (1) If $s_{ce} \in S_d$ then the relational verifier delivered a spurious counterexample, the dynamic slicer delivered an invalid dynamic slice, or the SDS algorithm computed a wrong simultaneous dynamic slice. (2) If P_{ce} contains no additional instruction compared to P_d, then $P_d \cup P_{ce} = P_d$ which means that P_d is a dynamic slice for s_{ce}. This implies that the relational verifier delivered a spurious counterexample. □

Theorem 3 guarantees that the CGS strategy adds at least one instruction back after each iteration. Thus, the number of calls of the relational verifier is linear in the number of program instructions. The SDS algorithm is needed for this theorem to hold. The validity of the slice computed with CGS, however, is guaranteed by the relational verifier. Thus, if the CGS algorithm computes the simple union of dynamic slices, the relational verifier my return a counterexample that it already provided in a previous CGS iteration. In this case the CGS algorithm needs to terminate and return the original program. Given the fact that computing the union of dynamic slices is much easier than computing the simultaneous dynamic slice, the user of the framework must make a choice between performance and completeness. Our implementation of CGS computes the union of dynamic slices.

The CGS strategy has the least number of calls to the relational verifier compared with the other strategies presented in this paper. Nevertheless, it comes with some disadvantages. First, the program needs to be executed at each iteration, which – depending on the analyzed program – can cause performance issues and for some inputs the program may not even terminate. Second, the CGS strategy is vulnerable to timeouts of the relational verifier. If a timeout occurs, then the strategy fails entirely and must return the original program as the slice candidate, while the BF and SSE strategies could continue their search for a valid slice candidate. Third, the precision of CGS depends on the precision of the dynamic slicing approach used in the candidate generation. Even though the used dynamic slicing approach can remove more statements than static syntactic slicing approaches, the dynamic slices it computes are still over-approximations.

5 Discussion

We start the discussion by reiterating the evaluation results [6] of the prototypical implementation of the framework consisting of the tool SemSlice [5,7], as shown in Table 1. For the evaluation, we used a collection of small but intricate examples (e.g., the example of Fig. 1 or a routine in which the same value is first added and then subtracted), each focusing on a particular challenge which cannot be handled by syntactic state of the art slicers. Some examples are taken from slicing literature [4,9,16,22,39]. The second column indicates the source of

Table 1. Evaluation

Original			BF			SSE			CGS		
Example	Source	#stmts	time (s)	#stmts	#calls	time (s)	#stmts	#calls	time (s)	#stmts	#calls
count_occurrence_error	self	50				13	42	11			
count_occurrence_result	self	50				16	44	13			
dead_code_after_ssa	[39]	4	< 1	2	4	< 1	2	4	< 1	2	1
dead_code_unused_variable	self	3	< 1	2	2	< 1	2	3	< 1	2	1
identity_not_modifying	[16]	8	< 1	3	3	< 1	7	5	< 1	6	1
identity_plus_minus_50	[4]	5	< 1	2	4	< 1	5	4	< 1	5	1
iflow_cyclic	[39]	18	62	14	2197	< 1	16	6	< 1	17	1
iflow_dynfamic_override	self	15	23	8	1298	< 1	11	8	< 1	12	1
iflow_endofloop (Fig. 1)	self	19	118	15	4065	< 1	16	7	< 1	18	2
intermediate	self	13	4	11	129	< 1	12	5	< 1	12	2
requires_path_sensitivity	[22]	20	647	16	26894	< 1	17	10	< 1	18	3
single_pass_removal	self	13	< 1	3	7	< 1	6	11	< 1	8	1
unchanged_over_itteration	self	20	29	9	932	1	15	14	< 1	20	2
unreachable_code_nested	self	10	< 1	2	1	< 1	9	1	< 1	4	1
whole_loop_removable	self	20	15	8	469	< 1	17	5	< 1	17	2

each example, the third the number of LLVM-IR statements in the program. For each slice candidate generation-strategy from Sect. 4 (BF, SSE, and CGS), the table lists the number of statements in the smallest slice found by SemSlice, the (wall) time needed by the tool, and the number of calls to the relational verifier. The experiments were conducted on a machine with an Intel Core I5-6600K CPU and 16 GB RAM. The exponential BF approach works satisfactorily fast on functions with up to 20 statements, and while it requires more time than the other approaches it computes more precise slices. For examples with less than 10 statements the brute-force approach takes less than one second. The other two approaches achieved slices of similar precision (to each other) and required less than one second for most examples. The evaluation shows that the framework can handle programs that require a large number of calls to the relational verifier, e.g. the program *requires_paths_sensitivity* with the BF strategy called the relational verifier almost 27000 times and took about 10 minutes to find the slice. The BF strategy serves as a *worst-case* scenario when using the slicing framework to automatically slice programs. Other strategies need fewer calls. For this example the other strategies were still able to remove some instructions with fewer less calls to the relational verifier and therefore they could scale to larger programs. Thus, the scalability of our slicing approach can be increased by using candidate generation strategies that do not call relational verifier often. Another way to ensure that our approach to slicing scales to large programs is to apply it to individual program functions (as opposed to applying it to the entire program). Our current prototypical implementation supports only a subset of the LLVM IR instruction set, which is the main reason we did not evaluate it on large, real-life programs.

Our slicing approach works on an intermediate representation language. This is beneficial for the implementation of the approach, as it does not need to handle all features of a modern high level programming language. However, one

of the uses for program slicing is to help the user debug and comprehend a program written in a high level language. It is possible to perform relational verification of such programs, the early version of LLRêve was in fact working on a simple *while* language in [14], LLVM-IR was later chosen [24] to increase the practicability of LLRêve. We believe the current framework can be adapted for slicing high level languages by either (1) attempting to translate back the IR slice to the high level language, or (2) by defining the slicing candidate in the high level language and then translating both the original program and the slice candidate into the IR and then using the extended relational verifier. For the first option we expect that only an over-approximation of the IR slice can be obtained by translating it back into the high level language, similar to what was done in [20]. As for the second solution, the CFGs of the original program and slice candidate in the IR may be so different that our approach would no longer be able to automatically find matching synchronization points. A solution to this would be to automatically annotate the original program and its slice candidate in the high level language, thus marking the synchronization points and using these marks in the IR translation. A further solution for supporting a high-level language would be to extend the work in [14] with the ideas of this paper. Thus, Definition 4 of a valid slice would need to be adapted for high-level programming languages and the weakest liberal precondition calculus from [14] would need to be extended such that it supports slicing in the case in which the criterion location is in the middle of the program. By working on the high-level programming language we would lose the advantages of working on an intermediate representation, i.e. relative language independence and existing support for various code optimizations, but our approach to slicing would become more suitable for program debugging and comprehension.

The IR language that we used to present our approach is not inter-procedural. While we could consider all programs as having been inlined beforehand, recursive procedures would not be supported. The relational verifier supports dealing with function calls using mutual function summaries [24] which abstract two matching function calls using coupling predicates. In general it is difficult to find matching function calls, but for checking the validity of slice candidates this can be done automatically, similar to finding matching synchronization points. Thus, our approach can be extended to support recursive functions; however the function calls themselves may not be removed, otherwise the mutual function summaries cannot be used.

In the semantics that we provided in Sect. 2 we assume that an error (e.g. a division by zero) causes the system to transition to the end state. An interesting question in the context of program slicing is whether instructions which may cause errors can be removed from the program. While some approaches (e.g. [33,34]) keep the error prone instructions in the slice, others (e.g. [29]) allow the removal of such statements but at the cost of a weaker soundness property (i.e., what constitutes a valid slice) which is nonetheless still useful in certain application scenarios such as software verification. With our slicing approach, we keep error prone instructions in the slice. However, because we take the semantics

of the program instructions into account, we can remove error prone instructions which will never cause an error, e.g. a division where the divisor will never be zero.

The completeness of our approach, i.e. whether a valid slice according to Definition 4 is deemed as such, is limited in practice by two factors. First, the relational verifier is required to automatically infer the coupling predicates needed to verify the validity of a slice candidate. The relational verifier works well when the needed coupling predicates are limited to linear arithmetics [26]. The second factor limiting completeness is the requirement that the original program and the slice candidate must run in lockstep. This is needed to ensure the mutual termination and that the criterion location is executed the same number of times. Thus, whereas we can remove instructions from inside a loop, we are not able to remove the loop itself (in our case the conditional jump instruction), even if it is empty – i.e. it loops over *skip* instructions.

6 Related Work

Static slicing is an active area of research and many approaches have been developed. We present those that are most similar to our work.

Assertion based slicing [4] also takes the semantics of the program into consideration. Program methods must be specified with a contract, which also represents the slicing criterion, i.e. statements are removed such that the reduced program still fulfills the contract. Unlike in our approach, loop invariants are required and only groups of instructions that are at consecutive program locations can be removed. This approach improves and combines older approaches [10,11], an implementation also exists [12]. The approach in [30] also uses a method's contract as the slicing criterion. However, the program parts that are deemed irrelevant are not removed, but replaced with an abstraction. Thus, the slice candidate over-approximates the behavior of the original program. If the contract is proved for the slice candidate, then it is also valid for the original program.

Path sensitive backward slicing [22] is another slicing approach that takes the program's semantics into consideration. The main idea is to symbolically execute the program and check the satisfiability of the path condition of every execution path. Only the satisfiable paths are used for computing the slice. The approach handles loops by using abstract interpretation to generate loop invariants, which can lead to an over-approximated description of the loop behavior. Thus, while the approach offers an increased precision when compared to syntactic approaches, it is not able slice the program in Fig. 1a. An implementation of this approach is available in the tool Tracer [23]. The idea of discarding dependencies that can only occur on infeasible program paths has also been explored in other works e.g. [9,36]. For these approaches, a compromise between the precision and scalability had to be found.

Abstract program slicing [17] is an approach which makes use of the program's semantics, however a different slicing criterion is used. Instead of preserving

those instructions that affect the exact values of the criterion variables at the criterion location, this approach preserves the statements that affect a property of the criterion variable. The properties pursued in this approach are whether the variables belong to a given abstract domain, e.g. the positive integers. Using abstract interpretation, for some operations the abstract domain of the output is known – provided the abstract domains of the inputs are also known. Thus some dependencies modeled in the PDG can be removed. This approach can generate slices which are not valid according to Definition 4.

The Frama-C framework [25] for software analysis provides components that support abstract interpretation and program slicing (based on program dependence graphs). Abstract interpretation can be used to improve the precision of the slicing component by identifying some infeasible branches. Abstract interpretation can automatically handle loops, but it does this by over-approximating their effects.

In [33] a different notion of semantic dependence between program statements is defined. In that work it is assumed that each node in the CFG of a program has an assigned function that represents the computation performed by that node. Thus, a statement s is semantically dependent on a statement s' if the interpretation of the function computed by s' affects the execution behavior of s. Consider a program that contains the instruction `assign x (x + 0)` followed by the criterion location and x as a criterion variable. According to the definition from [33] the assignment would be in the slice, because if the interpretation of the symbol + changes (e.g. to multiplication) then so would the value of x at the criterion location. In our approach, on the other hand, we consider the semantics of the program instructions to be fix, and can remove the statement from the slice, as it leaves the value of x unchanged.

Other, syntactic, slicing approaches have been surveyed in [41] and in [37], and a survey of dynamic slicing techniques can be found in [28].

7 Conclusion and Future Work

We extended a relational verification approach such that it can check whether a slice candidate is indeed a valid slice. Based on this, we built a framework for precise and automatic static slicing which consists of a candidate generation engine and the extended relational verifier. We presented three strategies to compute slice candidates, of which *counterexample guided slicing* is more sophisticated. It uses the counterexample provided by the relational verifier to refine the slice candidate with a dynamic slicer.

We plan to improve the precision of the slices by performing an additional analysis on empty loops to check whether they terminate. If this is the case, they can be safely removed. Furthermore, we plan to improve the performance of the relational verifier by using PDGs to simplify the programs that need to be checked for equivalence, using the fact that two programs with isomorphic PDGs are equivalent, as shown in [21]. We will also investigate how the results (e.g. coupling invariants) of the relational verifier can be reused when checking another slice candidate, constructed from the same original program.

References

1. LLVM language reference manual. https://llvm.org/docs/LangRef.html. Accessed 06 Feb 2019
2. Agrawal, H., Horgan, J.R.: Dynamic program slicing. In: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, PLDI 1990, pp. 246–256. ACM, New York (1990). https://doi.org/10.1145/93542.93576
3. Barraclough, R.W., et al.: A trajectory-based strict semantics for program slicing. Theoret. Comput. Sci. **411**(11), 1372–1386 (2010). https://doi.org/10.1016/j.tcs.2009.10.025
4. Barros, J.B., da Cruz, D., Henriques, P.R., Pinto, J.S.: Assertion-based slicing and slice graphs. Formal Aspects Comput. **24**(2), 217–248 (2012). https://doi.org/10.1007/s00165-011-0196-1
5. Beckert, B., Bormer, T., Gocht, S., Herda, M., Lentzsch, D., Ulbrich, M.: SemSlice: exploiting relational verification for automatic program slicing. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 312–319. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_20
6. Beckert, B., Bormer, T., Gocht, S., Herda, M., Lentzsch, D., Ulbrich, M.: Evaluation data of SemSlice (2019). https://doi.org/10.5281/zenodo.3334571
7. Beckert, B., Bormer, T., Gocht, S., Herda, M., Lentzsch, D., Ulbrich, M.: Implementation of the SemSlice tool (2019). https://doi.org/10.5281/zenodo.3334553
8. Binkley, D., Harman, M.: A survey of empirical results on program slicing. In: Advances in Computers, vol. 62, pp. 105–178. Elsevier (2004). https://doi.org/10.1016/S0065-2458(03)62003-6
9. Canfora, G., Cimitile, A., Lucia, A.D.: Conditioned program slicing. Inf. Softw. Technol. **40**(11–12), 595–607 (1998). https://doi.org/10.1016/S0950-5849(98)00086-X
10. Chung, I.S., Lee, W.K., Yoon, G.S., Kwon, Y.R.: Program slicing based on specification. In: Proceedings of the 2001 ACM Symposium on Applied Computing, SAC 2001, pp. 605–609. ACM, New York (2001). https://doi.org/10.1145/372202.372784
11. Comuzzi, J.J., Hart, J.M.: Program slicing using weakest preconditions. In: Gaudel, M.-C., Woodcock, J. (eds.) FME 1996. LNCS, vol. 1051, pp. 557–575. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-60973-3_107
12. da Cruz, D., Henriques, P.R., Pinto, J.S.: GamaSlicer: an online laboratory for program verification and analysis. In: Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications, LDTA 2010, pp. 3:1–3:8. ACM, New York (2010). https://doi.org/10.1145/1868281.1868284
13. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Relational verification through horn clause transformation. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 147–169. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_8
14. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE 2014, pp. 349–360. ACM (2014). https://doi.org/10.1145/2642937.2642987
15. 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). https://doi.org/10.1145/24039.24041

16. Field, J., Ramalingam, G., Tip, F.: Parametric program slicing. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1995, pp. 379–392. ACM, New York (1995). https://doi.org/10.1145/199448.199534
17. Halder, R., Cortesi, A.: Abstract program slicing on dependence condition graphs. Sci. Comput. Program. **78**(9), 1240–1263 (2013). https://doi.org/10.1016/j.scico.2012.05.007
18. Hall, R.J.: Automatic extraction of executable program subsets by simultaneous dynamic program slicing. Autom. Softw. Eng. **2**(1), 33–53 (1995). https://doi.org/10.1007/BF00873408
19. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Int. J. Inf. Secur. **8**(6), 399–422 (2009). https://doi.org/10.1007/s10207-009-0086-1
20. Herda, M., Tyszberowicz, S., Beckert, B.: Using dependence graphs to assist verification and testing of information-flow properties. In: Dubois, C., Wolff, B. (eds.) TAP 2018. LNCS, vol. 10889, pp. 83–102. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92994-1_5
21. Horwitz, S., Prins, J., Reps, T.: On the adequacy of program dependence graphs for representing programs. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1988, pp. 146–157. ACM, New York (1988). https://doi.org/10.1145/73560.73573
22. Jaffar, J., Murali, V., Navas, J.A., Santosa, A.E.: Path-sensitive backward slicing. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 231–247. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33125-1_17
23. Jaffar, J., Murali, V., Navas, J.A., Santosa, A.E.: TRACER: a symbolic execution tool for verification. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 758–766. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_61
24. Kiefer, M., Klebanov, V., Ulbrich, M.: Relational program reasoning using compiler IR - combining static verification and dynamic analysis. J. Autom. Reason. **60**(3), 337–363 (2017). https://doi.org/10.1007/s10817-017-9433-5
25. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: a software analysis perspective. Formal Aspects Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
26. Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification of pointer programs by predicate abstraction. Formal Methods Syst. Des. **52**(3), 229–259 (2018). https://doi.org/10.1007/s10703-017-0293-8
27. Korel, B., Laski, J.W.: Dynamic program slicing. Inf. Process. Lett. **29**(3), 155–163 (1988). https://doi.org/10.1016/0020-0190(88)90054-3
28. Korel, B., Rilling, J.: Dynamic program slicing methods. Inf. Softw. Technol. **40**(11–12), 647–659 (1998). https://doi.org/10.1016/S0950-5849(98)00089-5
29. Léchenet, J.-C., Kosmatov, N., Le Gall, P.: Cut branches before looking for bugs: sound verification on relaxed slices. In: Stevens, P., Wąsowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 179–196. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_11
30. Liu, T., Tyszberowicz, S., Herda, M., Beckert, B., Grahl, D., Taghdiri, M.: Computing specification-sensitive abstractions for program verification. In: Fränzle, M., Kapur, D., Zhan, N. (eds.) SETTA 2016. LNCS, vol. 9984, pp. 101–117. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47677-3_7

31. Lucia, A.D.: Program slicing: methods and applications. In: Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation, pp. 142–149, November 2001. https://doi.org/10.1109/SCAM.2001.972675
32. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
33. Podgurski, A., Clarke, L.A.: A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Trans. Softw. Eng. **16**(9), 965–979 (1990). https://doi.org/10.1109/32.58784
34. Ranganath, V.P., Amtoft, T., Banerjee, A., Hatcliff, J., Dwyer, M.B.: A new foundation for control dependence and slicing for modern program structures. ACM Trans. Program. Lang. Syst. **29**(5) (2007). https://doi.org/10.1145/1275497.1275502
35. Rümmer, P., Hojjat, H., Kuncak, V.: Disjunctive interpolants for horn-clause verification. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 347–363. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_24
36. Snelting, G., Robschink, T., Krinke, J.: Efficient path conditions in dependence graphs for software safety analysis. ACM Trans. Softw. Eng. Methodol. **15**(4), 410–457 (2006). https://doi.org/10.1145/1178625.1178628
37. Tip, F.: A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands (1994). https://www.franktip.org/pubs/jpl1995.pdf
38. Verdoolaege, S., Janssens, G., Bruynooghe, M.: Equivalence checking of static affine programs using widening to handle recurrences. ACM Trans. Program. Lang. Syst. **34**(3), 11:1–11:35 (2012). https://doi.org/10.1145/2362389.2362390
39. Ward, M.: Properties of slicing definitions. In: 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 23–32, September 2009. https://doi.org/10.1109/SCAM.2009.12
40. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, ICSE 1981, Piscataway, NJ, USA, pp. 439–449. IEEE Press (1981). http://dl.acm.org/citation.cfm?id=800078.802557
41. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. SIGSOFT Softw. Eng. Notes **30**(2), 1–36 (2005). https://doi.org/10.1145/1050849.1050865