Cut Branches Before Looking for Bugs: Sound Verification on Relaxed Slices

Jean-Christophe Léchenet[1,2], Nikolai Kosmatov[1], and Pascale Le Gall[2]

[1] CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr
[2] Laboratoire de Mathématiques et Informatique pour la Complexité et les Systèmes
CentraleSupélec, Université Paris-Saclay, 92295 Châtenay-Malabry France
firstname.lastname@centralesupelec.fr

Abstract. Program slicing can be used to reduce a given initial program to a smaller one (a *slice*) which preserves the behavior of the initial program with respect to a chosen criterion. Verification and validation (V&V) of software can become easier on slices, but require particular care in presence of errors or non-termination in order to avoid unsound results or a poor level of reduction in slices.
This article proposes a theoretical foundation for conducting V&V activities on a slice instead of the initial program. We introduce the notion of *relaxed slicing* that remains efficient even in presence of errors or non-termination, and establish an appropriate soundness property. It allows us to give a precise interpretation of verification results (absence or presence of errors) obtained for a slice in terms of the initial program. Our results have been proved in Coq.

1 Introduction

Context. Program slicing was initially introduced by Weiser [32, 33] as a technique allowing to decompose a given program into a simpler one, called a program slice, by analyzing its control and data flow. In the classic definition, a *(program) slice* is an executable program subset of the initial program whose behavior must be identical to a specified subset of the initial program's behavior. This specified behavior that should be preserved in the slice is called *slicing criterion*. A common slicing criterion is a program point l. For the purpose of this paper, we prefer this simple formulation to another criterion (l, V) where a set of variables V is also specified. Informally speaking, program slicing with respect to the criterion l should guarantee that any variable v at program point l takes the same value in the slice and in the original program.

Since Weiser's original work, many researchers have studied foundations of program slicing (e.g. [4–6, 8, 11, 14, 20, 26–28]). Numerous applications of slicing have been proposed, in particular, to program understanding, software maintenance, debugging, program integration and software metrics. Comprehensive surveys on program slicing can be found e.g. in [9, 29, 30, 35]. In recent classifications of program slicing, Weiser's original approach is called *static backward* *slicing* since it simplifies the program statically, for all possible executions at the same time, and traverses it backwards from the slicing criterion in order to keep those statements that can influence this criterion. Static backward slicing based on control and data dependencies is also the purpose of this work.

Goals and approach. Verification and Validation (V&V) can become easier on simpler programs after "cutting off irrelevant branches" [13, 15, 17, 22]. Our main goal is to address the following research question:

> **(RQ)** Can we soundly conduct V&V activities on slices instead of the initial program? In particular, if there are no errors in a program slice, what can be said about the initial program? And if an error is found in a program slice, does it necessarily occur in the initial program?

We consider errors determined by the current program state such as runtime errors (that can either interrupt the program or lead to an undefined behavior). We also consider a realistic setting of programs with potentially non-terminating loops, even if this non-termination is unintended. So we assume neither that all loops terminate, nor that all loops do not terminate, nor that we have a preliminary knowledge of which loops terminate and which loops do not.

Dealing with potential runtime errors and non-terminating loops is very important for realistic programs since their presence cannot be a priori excluded, especially during V&V activities. Although quite different at first glance, both situations have a common point: they can in some sense interrupt normal execution of the program preventing the following statements from being executed. Therefore, slicing away (that is, removing) potentially erroneous or non-terminating sub-programs from the slice can have an impact on soundness of program slicing.

While some aspects of **(RQ)** were discussed in previous papers, none of them provided a complete formal answer in the considered general setting (as we detail in Sec. 2 and 6 below). To satisfy the traditional soundness property, program slicing would require to consider additional dependencies of each statement on previous loops and error-prone statements. That would lead to inefficient (that is, too large) slices, where we would *systematically preserve all potentially erroneous or non-terminating statements* executed before the slicing criterion. Such slices would have very limited benefit for our purpose of performing V&V on slices instead of the initial program.

This work proposes *relaxed slicing,* where additional dependencies on previous (potentially) erroneous or non-terminating statements are not required. This approach leads to smaller slices, but needs a new soundness property. We state and prove a suitable soundness property using a trajectory-based semantics, and show how this result can justify V&V on slices by characterizing possible verification results on slices in terms of the initial program. The proof has been formalized in the Coq proof assistant [7] and is available in [1].

The contributions of this work include:

- a comprehensive analysis of issues arising for V&V on classic slices;
- the notion of relaxed slicing (Def. 6) for structured programs with possible errors and non-termination, that keeps fewer statements than it would be necessary to satisfy the classic soundness property of slicing;

– a new soundness property for relaxed slicing (Th. 1);
– a characterization of verification results, such as absence or presence of errors, obtained for a relaxed slice, in terms of the initial program, that constitutes a theoretical foundation for conducting V&V on slices (Th. 2, 3);
– a formalization and proof of our results in Coq.

Paper outline. Sec. 2 presents our motivation and illustrating examples. The considered language and its semantics are defined in Sec. 3. Sec. 4 defines the notion of relaxed slice and establishes its main soundness property. Next, Sec. 5 formalizes the relationship between the errors in the initial program and in a relaxed slice. Finally, Sec. 6 and 7 present the related work and the conclusion with some future work.

2 Motivation and Running Examples

Errors and assertions. We consider errors that are determined by the current program state[3] including runtime errors (division by zero, out-of-bounds array access, arithmetic overflows, out-of-bounds bit shifting, etc.). Some of these errors do not always interrupt program execution and can sometimes lead to an (even more dangerous) undefined behavior, such as reading or writing an arbitrary memory location after an out-of-bounds array access in C. Since we cannot take the risk to overlook some of these "silent runtime errors", we assume that all threatening statements are annotated with explicit assertions `assert(C)` placed before them, that interrupt the execution whenever the condition `C` is false. This assumption will be convenient for the formalization in the next sections: possible runtime errors will always occur in assertions. Such assertions can be generated syntactically (for example, by the RTE plugin of the FRAMA-C toolset [21] for C programs). For instance, line 10 in Fig. 1a prevents division by zero at line 11, while line 13 makes explicit a potential runtime error at line 14 if the array `a` is known to be of size `N`. In addition, the `assert(C)` keyword can be also used to express any additional user-defined properties on the current state.

Most previous applications of slicing to debugging used slices in order to *better understand an already detected error,* by analyzing a simpler program rather than a more complex one [8, 29, 30]. Our goal is quite different: to perform V&V on slices in order to discover yet unknown errors, or show their absence (cf. **(RQ)**). The interpretation of absence or presence of errors in a slice in terms of the initial program requires solid theoretical foundations.

Classic soundness property. Let p be a program, and q a slice of p w.r.t. a slicing criterion l. The classic soundness property of slicing (cf. [6, Def. 2.5] or [28, Slicing Th.]) can be informally stated as follows.

Property 1. Let σ be an input state of p. Suppose that p halts on σ. Then q halts on σ and the executions of p and q on σ agree after each statement preserved in the slice on the variables that appear in this statement.[4]

[3] Temporal errors (e.g. use-after-free in C) cannot be directly represented in this way.
[4] Formally, using the notation introduced hereafter in the paper (cf. Def. 8), their *projections* are equal: $\mathrm{Proj}_L(\mathcal{T}[\![p]\!]\sigma) = \mathrm{Proj}_L(\mathcal{T}[\![q]\!]\sigma)$.

```
 1 s1 = 0;
 2 s2 = 0;
 3 i = 0;
 4 while (i < N){
 5   assert (i < N);
 6   s1 = s1 + a[i];
 7   i = i + k;
 8 }
 9 j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0);
18 avg1 = s1 / N;
19 assert (N != 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```

(a)

```
 1 s1 = 0;
 2
 3 i = 0;
 4 while (i < N){
 5   assert (i < N);
 6   s1 = s1 + a[i];
 7   i = i + k;
 8 }
 9
10
11
12
13
14
15
16
17 assert (N != 0);
18 avg1 = s1 / N;
19
20
21
22
```

(b)

```
 1
 2 s2 = 0;
 3
 4
 5
 6
 7
 8
 9 j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17
18
19 assert (N != 0);
20 avg2 = s2 / N;
21
22
```

(c)

Fig. 1. (a) A program computing in two ways the average of elements of a given array `a` of size `N` whose only nonzero elements can be at indices $\{0, k, 2k, \dots\}$, and its two slices: **(b)** w.r.t. line 18, and **(c)** w.r.t. line 20.

This property was originally established for classic dependence-based slicing for programs without runtime errors and only for executions with terminating loops: nothing is guaranteed if p does not terminate normally on σ. Let us show why this property does not hold in presence of potential runtime errors or non-terminating loops.

Illustrating examples. Fig. 1a presents a simple (buggy) C-like program that takes as inputs an array `a` of length `N` and an integer `k` (with $0 \leqslant$ `k` $\leqslant 100$, $0 \leqslant$ `N` $\leqslant 100$), and computes in two different ways the average of the elements of `a`. We suppose that all variables and array elements are unsigned integers, and all elements of `a` whose index is not a multiple of `k` are zero, so it suffices to sum array elements over the indices multiples of `k` and to divide the sum by `N`. The sum is computed twice (in `s1` at lines 3–8 and in `s2` at lines 9–16), and the averages `avg1` and `avg2` are computed (lines 17–20) and compared (lines 21–22). We assume that necessary assertions with explicit guards (at lines $5, 10, 13, 17, 19$) are inserted to prevent runtime errors.

Fig. 1b shows a (classic dependence-based) slice of this program with respect to the statement at line 18. Intuitively, it contains only statements (at lines $1, 3, 4, 6, 7, 18$) that can influence the slicing criterion, i.e. the values of variables that appear at line 18 after its execution.[5] In addition, we keep the assertions to prevent potential errors in preserved statements. Similarly, Fig. 1c shows a slice with respect to line 20, again with protecting assertions.

[5] By formal definitions of Sec. 4, one easily checks that line 18 is data-dependent on line 6, that is in turn data-dependent on lines 1,3,7 and control-dependent on line 4.

Initial state	Inputs	**(a)**	**(b)**	**(c)**
σ_1	$k = 2,\ N = 5$	—	—	—
σ_2	$k = 2,\ N = 4$	↯ line 13	—	↯ line 13
σ_3	$k = 0,\ N = 4$	↺ line 4	↺ line 4	↯ line 10
σ_4	$k = 2,\ N = 0$	↯ line 13	↯ line 17	↯ line 13
σ_5	$k = 0,\ N = 0$	↯ line 10	↯ line 17	↯ line 10

Fig. 2. Errors (↯), non-termination (↺) and normal termination (—) of programs of Fig. 1 for some inputs.

Fig. 2 summarizes the behavior of the three programs of Fig. 1 on some test data. The elements of `a` do not matter here. Suppose we found an error at line 17 in slice **(b)** provoked by test datum σ_4. Program **(a)** does not contain the same error: it fails earlier, at line 13. We say that the error at line 17 in slice **(b)** is *hidden by the error* at line 13 of the initial program. Similarly, test datum σ_5 provokes an error at line 17 in slice **(b)** while this error is hidden by an error at line 10 in **(a)**. In fact, the error at line 17 cannot be reproduced on the initial program, so we say that it is *totally hidden* by other errors.

For slice **(c)**, detecting an error at line 10 on test datum σ_5 would allow us to observe the same error in **(a)**. However, if this error in slice **(c)** is also provoked by test datum σ_3, this test datum does not provoke any error in **(a)** because the loop at line 4 does not terminate. We say that this error is *(partially) hidden by a non-termination* of the loop at line 4.

These examples clearly show that Property 1 is not true in presence of errors or non-terminating loops for classic slices. Indeed, the executions of p and q may disagree at least for two reasons:

(i) a previously executed non-terminating loop not preserved in the slice, or
(ii) a previously executed failing statement not preserved in the slice.

Let us consider another example related to error-free programs. If we suppose that $0 < \texttt{k} \leqslant 100$, $0 < \texttt{N} \leqslant 100$, and replace `N/k` by `(N-1)/k` at line 11 of Fig. 1, neither slice contains any error. If we manage to verify the absence of errors on both slices, can we be sure that the initial program is error-free as well?

Bigger slices vs. weaker soundness property. One solution (adopted by [18, 25, 26]), cf. Sec. 6) proposes to ensure Property 1 even in presence of errors and potentially non-terminating loops by considering additional dependencies. This approach would basically lead to always preserving in the slice any (potentially non-terminating) loop or error-prone statement that can be executed before the slicing criterion. The resulting slices would be much bigger, and the benefit of performing V&V on slices would be very limited.

For instance, to ensure that the executions of program **(a)** and slice **(b)** activated by test datum σ_4 agree on all statements of slice **(b)**, line 13 should be preserved in slice **(b)**. That would result (by transitivity of dependencies) in keeping e.g. the loop at line 12 and lines 9–11 in slice **(b)** as well. Similarly, the loop at line 4 should be kept in slice **(c)** to avoid disagreeing executions for test datum σ_3. The slices can become much bigger in this approach.

In this paper we propose *relaxed slicing*, an alternative approach that does not require to keep all loops or error-prone statements that can be executed before the slicing criterion, but ensures a weaker soundness property. We demonstrate that the new soundness property is sufficient to justify V&V on slices instead of the initial program. In particular, we show that reasons **(i)** and **(ii)** above are the only possible reasons of a hidden error, and investigate when the absence of errors in slices implies the absence of errors in the initial program.

3 The Considered Language and its Semantics

Language. In this study, we consider a simple WHILE language (with integer variables, fixed-size arrays, pure expressions, conditionals, assertions and loops) that is representative for our formalization of slicing in presence of runtime errors and non-termination. The language is defined by the following grammar:

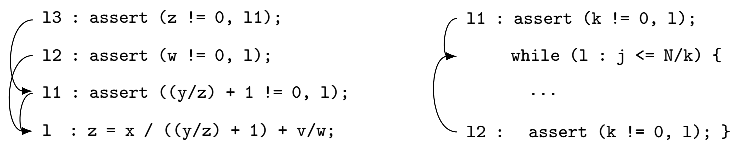
$$
\begin{array}{lll}
\mathit{Prog} & ::= & \mathit{Stmt}^* \\
\mathit{Stmt} & ::= & l : \texttt{skip} \mid \\
 & & l : x = e \mid \\
 & & \texttt{if}\ (l : b)\ \mathit{Prog}\ \texttt{else}\ \mathit{Prog} \mid \\
 & & \texttt{while}\ (l : b)\ \mathit{Prog} \mid \\
 & & l : \texttt{assert}\ (b, l')
\end{array}
$$

where l, l' denote labels, e an expression and b a boolean expression. A program (Prog) is a possibly empty list of statements (Stmt). The empty list is denoted λ, and the list separator is ";". We assume that the labels of any given program are distinct, so that a label uniquely identifies a statement. Assignments, conditions and loops have the usual semantics. As its name suggests, `skip` does nothing.

The assertion $\texttt{assert}(b, l')$ stops program execution in an error state (denoted ε) if b is false, otherwise execution continues normally. As said earlier, we assume that assertions are added to protect all threatening statements. The label l' allows us to associate the assertion with another statement that should be protected by the assertion (e.g. because it could provoke a runtime error). An assertion often protects the following line (like in Fig. 1, where the protected label is not indicated). Two simple cases however need more flexibility (cf. Fig. 3). Some assertions have to be themselves protected by assertions when they contain a threatening expression. Fig. 3a gives such an example where, instead of creating three assertions pointing to `l`, assertions `l1` and `l2` point to `l`, and assertion `l3` points to another assertion `l1`. Fig. 3b (inspired by the second loop of Fig. 1) shows how assertions with explicit labels can be used to protect a loop condition from a runtime error. The arrows in Fig. 3 indicate the protected statement.

Assertions can be also added by the user to check other properties than runtime errors. If the user does not need to indicate the protected statement, they can choose for l' either the label l of the assertion itself or any label not used elsewhere in the program. User-defined assertions should be also protected against errors by other assertions if necessary.

Semantics. Let p be a program. A program state is a mapping from variables to values. Let Σ denote the set of all valid states, and $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, where ε is



```
l3 : assert (z != 0, l1);

l2 : assert (w != 0, l);

l1 : assert ((y/z) + 1 != 0, l);

l  : z = x / ((y/z) + 1) + v/w;
```

(a) Chained assertions

```
l1 : assert (k != 0, l);

     while (l : j <= N/k) {

        ...

l2 :   assert (k != 0, l); }
```

(b) Loop condition

Fig. 3. Two special cases of assertions

the error state. Let σ be an initial state of p. The *trajectory* of the execution of p on σ, denoted $\mathcal{T}[\![p]\!]\sigma$, is the sequence of pairs $\langle(l_1,\sigma_1)\ldots(l_k,\sigma_k)\ldots\rangle$, where $l_1,\ldots,l_k,\ldots$ is the sequence of labels of the executed instructions, and σ_i is the state of the program *after* the execution of instruction l_i. $\mathcal{T}$ can be seen as a (partial) function

$$\mathcal{T} : Prog \to \Sigma \to Seq(L \times \Sigma_\varepsilon)$$

where $Seq(L \times \Sigma_\varepsilon)$ is the set of sequences of pairs $(l,\sigma) \in L \times \Sigma_\varepsilon$. Trajectories can be finite or (countably) infinite. A finite subsequence at the beginning of a trajectory T is called a *prefix* of T. The empty sequence is denoted $\langle\,\rangle$.

Let $\oplus$ be the concatenation operator over sequences. For a finite trajectory T, we denote by $LS_\sigma(T)$ the last state of T (i.e. the state component of its last element) if $T \neq \langle\,\rangle$, and σ otherwise. The definition of $T_1 \oplus T_2$ is standard if T_1 is finite. If T_1 is infinite or ends with the error state ε, then we set $T_1 \oplus T_2 = T_1$ for any T_2 (and even if T_2 is not well-defined, in other words, $\oplus$ performs lazy evaluation of its arguments).

We denote by $\mathcal{E}$ an evaluation function for expressions, that is standard and not detailed here. For any (pure) expression e and state $\sigma \in \Sigma$, $\mathcal{E}[\![e]\!]\sigma$ is the evaluation of expression e using σ to evaluate the variables present in e. The error state is only reached through a failed `assert`. Thanks to the assumption that all potentially failing statements are protected by assertions, we do not need to model errors in expressions or other statements: errors always occur in assertions. We also suppose for simplicity that all variables appearing in p are initialized in any initial state of p, that ensures the absence of expressions that cannot be evaluated due to an uninitialized variable. These assumptions slightly simplify the presentation without loss of generality for our purpose: loops and errors (in assertions) are present in the language.

Fig. 4 gives the inductive definition of $\mathcal{T}$ for any valid state $\sigma \in \Sigma$. The definitions for a loop and a conditional rely on the notation $(v \to T_1, T_2)$ also defined in Fig. 4. For any state σ, variable x and value v, $\sigma[x \leftarrow v]$ denotes σ overridden by the association $x \mapsto v$. Notice that in the definitions for a sequence and a loop, it is important that $\oplus$ does not evaluate the second parameter when the first trajectory is infinite or ends with the error state since the execution of the remaining part is not defined in this case. Thus ε can appear only once at the very end of a trajectory.

We illustrate these definitions on slice **(b)** of Fig. 1, denoted p_b. For every initial state σ of p_b and unsigned integer i, we define $\sigma^i = \sigma[s_1 \leftarrow (i \cdot a[0] \mod M_u)]$, where M_u denotes the maximal representable value of an unsigned integer. Then

$$\begin{array}{rcl}
\mathcal{T}[\![\lambda]\!]\sigma &=& \langle\,\rangle, \\
\mathcal{T}[\![s;\ p]\!]\sigma &=& \mathcal{T}[\![s]\!]\sigma \oplus \mathcal{T}[\![p]\!](LS_\sigma(\mathcal{T}[\![s]\!]\sigma)), \\
\mathcal{T}[\![l : \texttt{skip}]\!]\sigma &=& \langle (l,\sigma)\rangle, \\
\mathcal{T}[\![l : x = e]\!]\sigma &=& \langle (l,\sigma[x \leftarrow \mathcal{E}[\![e]\!]\sigma])\rangle, \\
\mathcal{T}[\![\texttt{if}\ (l:b)\ p\ \texttt{else}\ q]\!]\sigma &=& \langle (l,\sigma)\rangle \oplus (\mathcal{E}[\![b]\!]\sigma \to \mathcal{T}[\![p]\!]\sigma, \mathcal{T}[\![q]\!]\sigma), \\
\mathcal{T}[\![\texttt{while}\ (l:b)\ p]\!]\sigma &=& \langle (l,\sigma)\rangle \oplus (\mathcal{E}[\![b]\!]\sigma \to \\
&& \quad \mathcal{T}[\![p]\!]\sigma \oplus \mathcal{T}[\![\texttt{while}\ (l:b)\ p]\!](LS_\sigma(\mathcal{T}[\![p]\!]\sigma)),\ \langle\,\rangle), \\
\mathcal{T}[\![l : \texttt{assert}(b,l')]\!]\sigma &=& (\mathcal{E}[\![b]\!]\sigma \to \langle (l,\sigma)\rangle, \langle (l,\varepsilon)\rangle),
\end{array}$$

where for any trajectories T, T' and boolean value v, we define

$$(v \to T, T') = \begin{cases} T \text{ if } v = True, \\ T' \text{ if } v = False. \end{cases}$$

Fig. 4. Trajectory-based semantics of the language (for a valid state $\sigma \in \Sigma$)

the trajectory on σ_3 is infinite, while the trajectory on σ_5 leads to an error:

$$\mathcal{T}[\![p_b]\!]\sigma_3 = \langle (1,\sigma_3^0)(3,\sigma_3^0)(4,\sigma_3^0)(5,\sigma_3^0)(6,\sigma_3^1)(7,\sigma_3^1)(4,\sigma_3^1)(5,\sigma_3^1)(6,\sigma_3^2)(7,\sigma_3^2)\dots\rangle,$$
$$\mathcal{T}[\![p_b]\!]\sigma_5 = \langle (1,\sigma_5^0)(3,\sigma_5^0)(4,\sigma_5^0)(17,\varepsilon)\rangle.$$

4 Relaxed Program Slicing

4.1 Control and Data Dependences

Let $L(p)$ denote the set of labels of program p. Let us consider here a more general slicing criterion defined as a subset of labels $L_0 \subseteq L(p)$, and construct a slice with respect to all statements whose labels are in L_0. In particular, this generalization can be very useful when one wants to perform V&V on a slice with respect to several threatening statements. In this work we focus on dependence-based slicing, where a dependence relation $\mathcal{D} \subseteq L(p) \times L(p)$ is used to construct a slice. We write $l \xrightarrow[p]{\mathcal{D}} l'$ to indicate that l' depends on l according to $\mathcal{D}$, i.e. $(l,l') \in \mathcal{D}$. The definitions of control and data dependencies, denoted respectively $\mathcal{D}_c$ and $\mathcal{D}_d$, are standard, and given following [6].

Definition 1 (Control dependence $\mathcal{D}_c$). *The control dependencies in p are defined by* `if` *and* `while` *statements in p as follows:*

for any statement $\texttt{if}\ (l:b)\ q\ \texttt{else}\ r$ *and* $l' \in L(q) \cup L(r)$, *we define* $l \xrightarrow[p]{\mathcal{D}_c} l'$;

for any statement $\texttt{while}\ (l:b)\ q$ *and* $l' \in L(q)$, *we define* $l \xrightarrow[p]{\mathcal{D}_c} l'$.

For instance, in Fig. 1a, lines 5–7 are control-dependent on line 4, while lines 13–15 are control-dependent on line 12.

To define data dependence, we need the notion of (finite syntactic) paths. Let us denote again by $\oplus$ the concatenation of paths, extend $\oplus$ to sets of paths as the set of concatenations of their elements, and denote by "$*$" Kleene closure.

Definition 2 (Finite syntactic paths). *The set of finite syntactic paths $\mathcal{P}(p)$ of a program p is inductively defined as follows:*

$$\begin{aligned}
\mathcal{P}([\![\lambda]\!]) &= \{\lambda\}, & \mathcal{P}([\![\texttt{if}\ (l:b)\ p\ \texttt{else}\ q]\!]) &= \{l\} \oplus (\mathcal{P}(p) \cup \mathcal{P}(q)),\\
\mathcal{P}([\![s;\ p]\!]) &= \mathcal{P}(s) \oplus \mathcal{P}(p), & \mathcal{P}([\![\texttt{while}\ (l:b)\ p]\!]) &= (\{l\} \oplus \mathcal{P}(p))^* \oplus \{l\},\\
\mathcal{P}([\![l:\texttt{skip}]\!]) &= \{l\}, & \mathcal{P}([\![l:\texttt{assert}(b,l')]\!]) &= \{l\}.\\
\mathcal{P}([\![l:x=e]\!]) &= \{l\}, & &
\end{aligned}$$

For a given label l, let $\mathrm{def}(l)$ denote the set of variables defined at l (that is, $\mathrm{def}(l) = \{v\}$ if l is an assignment of variable v, and $\varnothing$ otherwise), and let $\mathrm{ref}(l)$ be the set of variables referenced at l. If l designates a conditional (or a loop) statement, $\mathrm{ref}(l)$ is the set of variables appearing in the condition; other variables appearing in its branches (or loop body) do not belong to $\mathrm{ref}(l)$. We denote by $\mathrm{used}(l)$ the set $\mathrm{def}(l) \cup \mathrm{ref}(l)$.

Definition 3 (Data dependence $\mathcal{D}_d$). *Let l and l' be labels of a program p. We say that there is a data dependency $l \xrightarrow[p]{\mathcal{D}_d} l'$ if $\mathrm{def}(l) \neq \varnothing$ and $\mathrm{def}(l) \subseteq \mathrm{ref}(l')$ and there exists a path $\pi = \pi_1 l \pi_2 l' \pi_3 \in \mathcal{P}(p)$ such that for all $l'' \in \pi_2$, $\mathrm{def}(l'') \neq \mathrm{def}(l)$. Each π_i may be empty.*

For instance, in Fig. 1b, line 18 is data-dependent on line 1 (with $\pi = 1, 3, 4, 17, 18$) and on line 6 (with $\pi = 1, 3, 4, 5, 6, 7, 4, 17, 18$), while line 6 is data-dependent on lines 1, 3, 6 and 7.

A slice of p is expected to be a *quotient* of p, that is, a well-formed program obtained from p by removing zero, one or more statements. A quotient can be identified by the set of labels of preserved statements. Notice that when a conditional (or a loop) statement is removed, it is removed with all statements of its both branches (or its loop body) to preserve the structure of the initial program in the quotient.

Given a dependence relation $\mathcal{D}$ and $L_0 \subseteq L(P)$, the slice based on $\mathcal{D}$ w.r.t. L_0 will be also identified by the set of labels of preserved statements. The following lemma justifies the correctness of the definitions of slices given hereafter. We denote by $\mathcal{D}^*$ the reflexive transitive closure of $\mathcal{D}$, and by $(\mathcal{D}^*)^{-1}(L_0)$ the set of all labels $l' \in L(p)$ such that there exists $l \in L_0$ with $l' \xrightarrow[p]{\mathcal{D}^*} l$.

Lemma 1. *Let $L_0 \subseteq L(P)$. If $\mathcal{D}$ is a dependence relation on p such that $\mathcal{D}_c \subseteq \mathcal{D}$, then $(\mathcal{D}^*)^{-1}(L_0)$ is the set of labels of a (uniquely defined) quotient of p.*

Lemma 1 can be easily proven by structural induction. It allows us to define a slice as the set of statements on which the statements in L_0 are (directly or indirectly) dependent.

Definition 4 (Dependence-based slice). *Let $\mathcal{D}$ be a dependence relation on p such that $\mathcal{D}_c \subseteq \mathcal{D}$, and $L_0 \subseteq L(P)$. A* dependence-based slice *of p based on $\mathcal{D}$ with respect to L_0 is the quotient of p whose set of labels is $(\mathcal{D}^*)^{-1}(L_0)$. A* classic dependence-based slice *of p with respect to L_0 is based on $\mathcal{D} = \mathcal{D}_c \cup \mathcal{D}_d$.*

4.2 Assertion Dependence and Relaxed Slices

Soundness of classic slicing for programs without runtime errors or non-terminating loops can be expressed by Property 1 in Sec. 2. As we illustrated, to generalize this property in presence of runtime errors and for non-terminating executions one would need to add additional dependencies and systematically preserve in the slice all potentially erroneous or non-terminating statements executed before (a statement of) the slicing criterion. We propose here an alternative approach, called *relaxed slicing*, where only one additional dependency type is considered.

Definition 5 (Assertion dependence $\mathcal{D}_a$). *For every assertion l :* `assert` (b, l') *in p with $l, l' \in L(p)$, we define an assertion dependency $l \xrightarrow[p]{\mathcal{D}_a} l'$.*

Definition 6 (Relaxed slice). *A* relaxed slice *of p with respect to L_0 is the quotient of p whose set of labels is $(\mathcal{D}^*)^{-1}(L_0)$, where $\mathcal{D} = \mathcal{D}_c \cup \mathcal{D}_d \cup \mathcal{D}_a$.*

For instance, in Fig. 1a, there would be an assertion dependence of each threatening statement on the corresponding protecting assertion (written on the previous line). Therefore both slices **(b)** and **(c)** of Fig. 1 (in which we artificially preserved assertions in Sec. 2) are in fact relaxed slices where assertions are naturally preserved thanks to the assertion dependence.

Assertion dependence brings two benefits. It ensures that a potentially threatening instruction is never kept without its protecting assertion. At the same time, an assertion can be preserved without its protected statement, that is quite useful for V&V that focus on assertions: slicing w.r.t. assertions may produce smaller slices if we do not need the whole threatening statement. For example, a relaxed slice w.r.t. the assertion at line 17 would contain only this unique line.

Notice that a relaxed slice does not require to include potentially erroneous or non-terminating statements that can prevent the slicing criterion from being executed (like in [18, 25, 26]). For example, slice **(b)** does not include the potential error at line 13, and slice **(c)** does not include the loop of line 4.

4.3 Soundness of Relaxed Slicing

We cannot directly compare the trajectory of the original program with a slice, since it may refer to statements and variables not preserved in the slice. We use projections of trajectories that reduce them to selected labels and variables.

Definition 7 (Projection of a state). *The projection of a state σ to a set of variables V, denoted $\sigma{\downarrow}V$, is the restriction of σ to V if $\sigma \neq \varepsilon$, and ε otherwise.*

Definition 8 (Projection of a trajectory). *The projection of a one-element sequence $\langle(l,\sigma)\rangle$ to a set of labels L, denoted $\langle(l,\sigma)\rangle{\downarrow}L$, is defined as follows:*

$$\langle(l,\sigma)\rangle{\downarrow}L = \begin{cases} \langle(l, \sigma{\downarrow}\,\mathrm{used}(l))\rangle & \text{if } l \in L, \\ \langle\,\rangle & \text{otherwise.} \end{cases}$$

The projection of a trajectory $T = \langle(l_1,\sigma_1)\ldots(l_k,\sigma_k)\ldots\rangle$ to L, denoted $\mathrm{Proj}_L(T)$, is defined element-wise: $\mathrm{Proj}_L(T) = \langle(l_1,\sigma_1)\rangle{\downarrow}L \oplus \ldots \oplus \langle(l_k,\sigma_k)\rangle{\downarrow}L \oplus \ldots$.

We can now state and prove the soundness property of relaxed slices.

Theorem 1 (Soundness of a relaxed slice). *Let $L_0 \subseteq L(p)$ be a slicing criterion of program p. Let q be the relaxed slice of p with respect to L_0, and $L = L(q)$ the set of labels preserved in q. Then for any initial state $\sigma \in \Sigma$ of p and finite prefix T of $\mathcal{T}[\![p]\!]\sigma$, there exists a prefix T' of $\mathcal{T}[\![q]\!]\sigma$, such that:*

$$\mathrm{Proj}_L(T) = \mathrm{Proj}_L(T')$$

Moreover, if p terminates without error on σ, $\mathcal{T}[\![p]\!]\sigma$ and $\mathcal{T}[\![q]\!]\sigma$ are finite, and

$$\mathrm{Proj}_L(\mathcal{T}[\![p]\!]\sigma) = \mathrm{Proj}_L(\mathcal{T}[\![q]\!]\sigma)$$

Proof. Let $\sigma \in \Sigma$, $\mathcal{T}[\![p]\!]\sigma = \langle (l_1, \sigma_1)(l_2, \sigma_2)\dots\rangle$, and $\mathcal{T}[\![q]\!]\sigma = \langle (l'_1, \sigma'_1)(l'_2, \sigma'_2)\dots\rangle$. Let $T = \langle (l_1, \sigma_1)\dots(l_i, \sigma_i)\rangle$ be a finite prefix of $\mathcal{T}[\![p]\!]\sigma$. By Def. 8, the projections of $\mathcal{T}[\![q]\!]\sigma$ and T to $L = L(q)$ have the following form

$$\begin{aligned} \mathrm{Proj}_L(\mathcal{T}[\![q]\!]\sigma) &= \langle\, (l'_1, \sigma'_1{\downarrow}\,\mathrm{used}(l'_1))(l'_2, \sigma'_2{\downarrow}\,\mathrm{used}(l'_2))\dots\rangle, \\ \mathrm{Proj}_L(T) &= \langle\, (l_{f(1)}, \sigma_{f(1)}{\downarrow}\,\mathrm{used}(l_{f(1)}))\ \dots\ (l_{f(j)}, \sigma_{f(j)}{\downarrow}\,\mathrm{used}(l_{f(j)}))\,\rangle, \end{aligned}$$

where $j \leqslant i$ and f is a strictly increasing function.

Let us denote by k the greatest natural number such that $k \leqslant j$ and such that the prefix of $\mathcal{T}[\![q]\!]\sigma$ of length k exists and satisfies $(\mathrm{Proj}_L(T))^k = \mathrm{Proj}_L((\mathcal{T}[\![q]\!]\sigma)^k)$, where we denote by U^k the prefix of length k for any trajectory U. Let $T' = \langle (l'_1, \sigma'_1)\dots(l'_k, \sigma'_k)\rangle$ be the prefix $(\mathcal{T}[\![q]\!]\sigma)^k$. By Def. 8 we have

$$\mathrm{Proj}_L(T') = \langle\, (l'_1, \sigma'_1{\downarrow}\,\mathrm{used}(l'_1))\ \dots\ (l'_k, \sigma'_k{\downarrow}\,\mathrm{used}(l'_k))\,\rangle.$$

Since $(\mathrm{Proj}_L(T))^k = \mathrm{Proj}_L(T')$, for any $m = 1, 2, \dots, k$ we have $l'_m = l_{f(m)}$ and $\sigma'_m{\downarrow}\,\mathrm{used}(l'_m) = \sigma_{f(m)}{\downarrow}\,\mathrm{used}(l_{f(m)})$. Set $\sigma_0 = \sigma'_0 = \sigma$.

Let us prove that $k = j$. We reason by contradiction and assume that $k < j$. By maximality of k, there can be three different cases:

1. $\mathcal{T}[\![q]\!]\sigma$ is of size k, or
2. l'_{k+1} exists, but $l'_{k+1} \neq l_{f(k+1)}$, or
3. l'_{k+1} exists, $l'_{k+1} = l_{f(k+1)}$, but $\sigma'_{k+1}{\downarrow}\,\mathrm{used}(l'_{k+1}) \neq \sigma_{f(k+1)}{\downarrow}\,\mathrm{used}(l_{f(k+1)})$.

Since $l'_k = l_{f(k)}$, cases 1 and 2 can be only due to a diverging evaluation of a control flow statement (i.e. `if`, `while` or `assert`) situated in the execution of p between $l_{f(k)}$ and $l_{f(k+1)-1}$. If such a statement occurs at label $l'_k = l_{f(k)}$, its condition would be evaluated identically in both executions since $\sigma'_k{\downarrow}\,\mathrm{used}(l'_k) = \sigma_{f(k)}{\downarrow}\,\mathrm{used}(l_{f(k)})$. The first non-equal label $l_{f(k+1)}$ cannot be part of the body of some non-preserved `if` or `while` statement between $l_{f(k)} + 1$ and $l_{f(k+1)-1}$ in p by definition of control dependence (cf. Def. 1). Finally, the divergence cannot be due to an `assert` in p between $l_{f(k)+1}$ and $l_{f(k+1)-1}$ either, because a passed `assert` has no effect, while a failing `assert` would make it impossible to reach $l_{f(k+1)}$ in p. Thus a divergence leading to cases 1 and 2 is impossible.

In case 3, the key idea is to remark that $\sigma'_k{\downarrow}\,\mathrm{ref}(l'_{k+1}) = \sigma_{f(k+1)-1}{\downarrow}\,\mathrm{ref}(l_{f(k+1)})$. Indeed, assume that there is a variable $v \in \mathrm{ref}(l'_{k+1}) = \mathrm{ref}(l_{f(k+1)})$ such that

$\sigma'_k(v) \neq \sigma_{f(k+1)-1}(v)$. The last assignment to v in the execution of p before its usage at $l_{f(k+1)}$ must be preserved in q because of data dependence (cf. Def. 3), so it has a label $l'_u = l_{f(u)}$ for some $1 \leqslant u \leqslant k$. By definition of k, the state projections after this statement were equal: $\sigma'_u \downarrow \text{used}(l'_u) = \sigma_{f(u)} \downarrow \text{used}(l_{f(u)})$, so the last values assigned to v before its usage at $l_{f(k+1)}$ were equal, that contradicts the assumption $\sigma'_k(v) \neq \sigma_{f(k+1)-1}(v)$. This shows that all variables referenced in $l_{f(k+1)}$ have the same values, so the resulting states cannot differ, and case 3 is not possible either. Therefore $k = j$, and T' satisfies $\text{Proj}_L(T) = \text{Proj}_L(T')$.

If p terminates without error on σ, by the first part of the theorem we have a prefix T' of $\mathcal{T}[\![q]\!]\sigma$ such that $\text{Proj}_L(\mathcal{T}[\![p]\!]\sigma) = \text{Proj}_L(T')$. If T' is a strict prefix of $\mathcal{T}[\![q]\!]\sigma$, this means as before that a control flow statement executed in p causes the divergence of the two trajectories. By hypothesis, there are no failing assertions in the execution of p, therefore it is due to an `if` or a `while`. By the same reasoning as in cases 1, 2 above we show that its condition must be evaluated in the same way in both trajectories and cannot lead to a divergence. Therefore, $T' = \mathcal{T}[\![q]\!]\sigma$. □

5 Verification on Relaxed Slices

In this section, we show how the absence and the presence of errors in relaxed slices can be soundly interpreted in terms of the initial program.

Lemma 2. *Let q be a relaxed slice of p and $\sigma \in \Sigma$ an initial state of p. If the preserved assertions do not fail in the execution of q on σ, they do not fail in the execution of p on σ either.*

Proof. Let us show the contrapositive. Assume that $\mathcal{T}[\![p]\!]\sigma$ ends with (l, ε) where $l \in L(q)$ is a preserved assertion. Let $L = L(q)$. From Th. 1 applied to $T = \mathcal{T}[\![p]\!]\sigma$, it follows that there exists a finite prefix T' of $\mathcal{T}[\![q]\!]\sigma$ such that $\text{Proj}_L(T) = \text{Proj}_L(T')$. The last state of $\text{Proj}_L(T')$ is ε, therefore the last state of T' is ε too. It means that ε appears in $\mathcal{T}[\![q]\!]\sigma$, and by definition of semantics (cf. Sec. 3) this is possible only if ε is its last state. Therefore $\mathcal{T}[\![q]\!]\sigma$ ends with (l, ε) as well. □

The following theorem and corollary immediately follow from Lemma 2.

Theorem 2. *Let q be a relaxed slice of p. If all assertions contained in q never fail, then the corresponding assertions in p never fail either.*

Corollary 1. *Let $q_1, \ldots, q_n$ be relaxed slices of p such that each assertion in p is preserved in at least one of the q_i. If no assertion in any q_i fails, then no assertion fails in p.*

The last result justifies the detection of errors in a relaxed slice.

Theorem 3. *Let q be a relaxed slice of p and $\sigma \in \Sigma$ an initial state of p. We assume that $\mathcal{T}[\![q]\!]\sigma$ ends with an error state. Then one of the following cases holds for p:*

(†) *$\mathcal{T}[\![p]\!]\sigma$ ends with an error at the same label, or*
(††) *$\mathcal{T}[\![p]\!]\sigma$ ends with an error at a label not preserved in q, or*
(†††) *$\mathcal{T}[\![p]\!]\sigma$ is infinite.*

Proof. Let $L = L(q)$ and assume that $\mathcal{T}[\![q]\!]\sigma$ ends with (l, ε) for some preserved assertion at label $l \in L$. We reason by contradiction and assume that $\mathcal{T}[\![p]\!]\sigma$ does not satisfy any of the three cases. Then two cases are possible.

First, $\mathcal{T}[\![p]\!]\sigma$ ends with (l', ε) for another preserved assertion at label $l' \in L$ (with $l' \neq l$). Then reasoning as in the proof of Lemma 2 we show that $\mathcal{T}[\![q]\!]\sigma$ ends with (l', ε) as well, that contradicts $l' \neq l$.

Second, $\mathcal{T}[\![p]\!]\sigma$ is finite without error. Then the second part of Th. 1 can be applied and thus $\mathrm{Proj}_L(\mathcal{T}[\![p]\!]\sigma) = \mathrm{Proj}_L(\mathcal{T}[\![q]\!]\sigma)$. This is contradictory since $\mathcal{T}[\![q]\!]\sigma$ contains an error (at label $l \in L$) and $\mathcal{T}[\![p]\!]\sigma$ does not. □

For instance, consider the example of Fig. 1 with $0 < \texttt{k} \leqslant 100$, $0 < \texttt{N} \leqslant 100$. In this case we can prove that slice **(b)** does not contain any error, thus we can deduce by Th. 2 that the assertions at lines 5 and 17 (preserved in slice **(b)**) never fail in the initial program either. If in addition we replace `N/k` by `(N-1)/k` at line 11 of Fig. 1, we can show that neither of the two slices of Fig. 1 contains any error. Since these slices cover all assertions, we can deduce by Cor. 1 that the initial program is error-free.

Th. 3 shows that despite the fact that an error detected in q does not necessary appear in p, the detection of errors on q has a precise interpretation. It can be particularly meaningful for programs supposed to terminate, for which a non-termination within some time τ is seen as an anomaly. In this case, detection of errors in a slice is sound in the sense that if an error is found in q for initial state σ, there is an anomaly (same or earlier error, or non-termination within time τ) in p whose type can be easily determined by running p on σ.

It can be noticed that a result similar to Th. 3 can be established for non-termination: if $\mathcal{T}[\![q]\!]\sigma$ is infinite, then either (††) or (†††) holds for p.

6 Related Work

Weiser [34] introduced the basics of intraprocedural and interprocedural static slicing. A thorough survey provided in [30] explores both static and dynamic slicing and compares the different approaches. It also lists the application areas of program slicing. More recent surveys can be found at [9, 29, 35]. Foundations of program slicing have been studied e.g. in [4–6, 8, 11, 14, 20, 26–28]. This section presents a selection of works that are most closely related to the present paper.

Debugging and dynamic slicing. Program debugging and testing are traditional application domains of slicing (e.g. [2, 19, 33]) where it can be used to better understand an already detected error, to prioritize test cases (e.g. in regression testing), simplify a program before testing, etc. In particular, dynamic slicing [8] is used to simplify the program for a given (e.g. erroneous) execution. However, theoretical foundations of applying V&V on slices instead of the initial program (like in [13, 22]) in presence of errors and non-termination, that constitute the main purpose of this work, have been only partially studied.

Slicing and non-terminating programs. A few works tried to propose a semantics preserved by classic slicing even in presence of non-termination. Among them, we can cite the lazy semantics of [11], and the transfinite one of [16], improved by [24]. Another semantics proposed in [6] has several improvements compared to the previous ones: it is intuitive and substitutive. Despite the elegance of these proposals, they turn out to be unsuitable for our purpose because they consider non-existing trajectories, that are not adapted to V&V techniques, for example, based on path-oriented testing like in [13, 15].

[26] provides foundations for the slicing of modern programs, i.e. programs with exceptions and potentially infinite loops, represented by control flow graphs (CFG) and program dependence graphs (PDG). Their work gives two definitions of control dependence, non-termination sensitive and non-termination insensitive, corresponding respectively to the weak and strong control dependences of [25] and further generalized for any finite directed graph in [14]. [26] also establishes the soundness of classic slicing with non-termination sensitive control dependence in terms of weak bisimulation, more adapted to deal with infinite executions. Their approach requires to preserve all loops, that results in much bigger slices than in relaxed slicing.

[4] establishes a soundness property for non-termination insensitive control dependence in terms of simulation. [5] describes program slicing for arbitrary control flow. [5] and [4] state that an execution in the initial program can be a prefix of that in a slice, without carefully formalizing runtime errors. Our work establishes a similar property, and in addition performs a complete formalization of slicing in presence of errors *and* non-termination, explicitly formalizes errors by assertions and deduces several results on performing V&V on slices.

Slicing in presence of errors. [18] notes that classic algorithms only preserve a lazy semantics. To obtain correct slices with respect to a strict semantics, it proposes to preserve all potentially erroneous statements through adding pseudo-variables in the $\mathrm{def}(l)$ and $\mathrm{ref}(l)$ sets of all potentially erroneous statements l. Our approach is more fine-grained in the sense that we can independently select assertions to be preserved in the slice and to be considered by V&V on this slice. This benefit comes from our dedicated formalization of errors with assertions and a rigorous proof of soundness using a trajectory-based semantics. In addition, we make a formal link about the presence or the absence of errors in the program and its slices. [17] uses program slicing as well as meaning-preserving transformations to analyze a property of a program not captured by its own variables. For that, it adds variables and assignments in the same idea as our assertions. [3] extends data and control dependences for Java program with exceptions. In both papers, no formal justification is given.

Certified slicing. The ideas developed in [4, 26] were applied in [10, 31]. [31] builds a framework in Isabelle/HOL to formally prove a slicing defined in terms of graphs, therefore language-independent. [10] proposes an unproven but efficient slice calculator for an intermediate language of the CompCert C compiler [23], as well as a certified slice validator and a slice builder written in Coq [7].

The modeling of errors and the soundness of V&V on slices were not specifically addressed in these works.

To the best of our knowledge, the present work is the first complete formalization of program slicing for structured programs in presence of errors and non-termination. Moreover, it has been formalized in the Coq proof assistant on a representative structured language, that provides a certified program slicer and justifies conducting V&V on slices instead of the initial program.

7 Conclusion

In many domains, modern software has become very complex and increasingly critical. This explains both the growing efforts on verification and validation (V&V) and, in many cases, the difficulties to analyze the whole program. We revisit the usage of program slicing to simplify the program before V&V, and study how it can be performed in a sound way in presence of possible runtime errors (that we model by assertions) and non-terminating loops. Rather than preserving more statements in a slice in order to satisfy the classic soundness property (stating an equality of whole trajectory projections), we define smaller, *relaxed slices* where only assertions are kept in addition to classic control and data dependences, and prove a weaker soundness property (relating prefixes of trajectory projections). It allows us to formally justify V&V on relaxed slices instead of the initial program, and to give a complete sound interpretation of presence or absence of errors in slices. First experiments with SANTE [12, 13], where all-path testing is used on relaxed slices to confirm or invalidate alarms initially detected by value analysis, show that using relaxed slicing allowed to reduce the program in average by 51% (going up to 97% for some examples) and accelerated V&V in average by 43%.

The present study has been formalized in Coq for a representative programming language with assertions and loops, and the results of this paper (as well as many helpful additional lemmas on dependencies and slices) were proved in Coq, providing a certified correct-by-construction slicer for the considered language [1]. This Coq formalization represents an effort of 8 person-months of intensive Coq development resulting in more than 10,000 lines of Coq code.

Future work includes a generalization to a wider class of errors, an extension to a realistic programming language and a certification of a complete verification technique relying on program slicing. Another research direction is to precisely measure the reduction rate and benefits for V&V of relaxed slicing compared to slicing approaches systematically introducing dependencies on previous loops and erroneous statements. In an ongoing work in DEWI project, we apply relaxed slicing for verification of protocols of wireless sensor networks.

Acknowledgments. Part of the research work leading to these results has received funding for DEWI project (www.dewi-project.eu) from the ARTEMIS Joint Undertaking under grant agreement No. 621353. The authors thank Omar Chebaro, Alain Giorgetti and Jacques Julliand for many fruitful discussions and earlier work that lead to the initial ideas of this paper. Many thanks to the anonymous reviewers for lots of very helpful suggestions.

References

1. Formalization of relaxed slicing (2016), `http://perso.ecp.fr/~lechenetjc/slicing/`
2. Agrawal, H., DeMillo, R.A., Spafford, E.H.: Debugging with dynamic slicing and backtracking. Softw., Pract. Exper. 23(6), 589–616 (1993)
3. Allen, M., Horwitz, S.: Slicing java programs that throw and catch exceptions. In: PEPM 2003. pp. 44–54 (2003)
4. Amtoft, T.: Slicing for modern program structures: a theory for eliminating irrelevant loops. Inf. Process. Lett. 106(2), 45–51 (2008)
5. Ball, T., Horwitz, S.: Slicing programs with arbitrary control-flow. In: AADEBUG 1993 (1993)
6. Barraclough, R.W., Binkley, D., Danicic, S., Harman, M., Hierons, R.M., Kiss, A., Laurence, M., Ouarbya, L.: A trajectory-based strict semantics for program slicing. Theor. Comp. Sci. 411(11–13), 1372–1386 (2010)
7. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Springer (2004)
8. Binkley, D., Danicic, S., Gyimóthy, T., Harman, M., Kiss, Á., Korel, B.: Theoretical foundations of dynamic program slicing. Theor. Comput. Sci. 360(1-3), 23–41 (2006)
9. Binkley, D., Harman, M.: A survey of empirical results on program slicing. Advances in Computers 62, 105–178 (2004)
10. Blazy, S., Maroneze, A., Pichardie, D.: Verified validation of program slicing. In: CPP 2015. pp. 109–117 (2015)
11. Cartwright, R., Felleisen, M.: The semantics of program dependence. In: PLDI 1989
12. Chebaro, O., Cuoq, P., Kosmatov, N., Marre, B., Pacalet, A., Williams, N., Yakobowski, B.: Behind the scenes in SANTE: a combination of static and dynamic analyses. Autom. Softw. Eng. 21(1), 107–143 (2014)
13. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: SAC 2012
14. Danicic, S., Barraclough, R.W., Harman, M., Howroyd, J., Kiss, Á., Laurence, M.R.: A unifying theory of control dependence and its application to arbitrary program structures. Theor. Comput. Sci. 412(49), 6809–6842 (2011)
15. Ge, X., Taneja, K., Xie, T., Tillmann, N.: DyTa: dynamic symbolic execution guided with static verification results. In: the 33rd International Conference on Software Engineering (ICSE 2011). pp. 992–994. ACM (2011)
16. Giacobazzi, R., Mastroeni, I.: Non-standard semantics for program slicing. Higher-Order and Symbolic Computation 16(4), 297–339 (2003)
17. Harman, M., Danicic, S.: Using program slicing to simplify testing. Softw. Test., Verif. Reliab. 5(3), 143–162 (1995)
18. Harman, M., Simpson, D., Danicic, S.: Slicing programs in the presence of errors. Formal Aspects of Computing 8(4), 490–497 (1996)
19. Hierons, R.M., Harman, M., Danicic, S.: Using program slicing to assist in the detection of equivalent mutants. Softw. Test., Verif. Reliab. 9(4), 233–262 (1999)
20. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: PLDI 1988
21. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. 27(3), 573–609 (2015)

22. Kiss, B., Kosmatov, N., Pariente, D., Puccetti, A.: Combining static and dynamic analyses for vulnerability detection: Illustration on Heartbleed. In: HVC 2015
23. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
24. Nestra, H.: Transfinite semantics in the form of greatest fixpoint. J. Log. Algebr. Program. 78(7), 573–592 (2009)
25. Podgurski, A., Clarke, L.A.: A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Trans. Software Eng. 16(9), 965–979 (1990)
26. 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)
27. Reps, T.W., Yang, W.: The semantics of program slicing and program integration. In: TAPSOFT 1989
28. Reps, T.W., Yang, W.: The semantics of program slicing. Tech. rep., Univ. of Wisconsin (1988)
29. Silva, J.: A vocabulary of program slicing-based techniques. ACM Comput. Surv. 44(3), 12 (2012)
30. Tip, F.: A survey of program slicing techniques. J. Prog. Lang. 3(3) (1995)
31. Wasserrab, D.: From formal semantics to verified slicing: a modular framework with applications in language based security. Ph.D. thesis, Karlsruhe Inst. of Techn. (2011)
32. Weiser, M.: Program slicing. In: ICSE 1981
33. Weiser, M.: Programmers use slices when debugging. Commun. ACM 25(7), 446–452 (1982)
34. Weiser, M.: Program slicing. IEEE Trans. Software Eng. 10(4), 352–357 (1984)
35. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. ACM SIGSOFT Software Engineering Notes 30(2), 1–36 (2005)