Under consideration for publication in Formal Aspects of Computing

Cut Branches before Looking for Bugs: Certifiably Sound Verification on Relaxed Slices

Jean-Christophe Léchenet[1,2]

Nikolai Kosmatov[1]

Pascale Le Gall[2]

[1]CEA, LIST, Software Reliability and Security Laboratory, PC 174, 91191 Gif-sur-Yvette France
[2]Laboratoire de Mathématiques et Informatique pour la Complexité et les Systèmes,
CentraleSupélec, Université Paris-Saclay, 92295 Châtenay-Malabry France

Abstract. Program slicing can be used to reduce a given initial program to a smaller one (a *slice*) that 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 the presence of errors or non-termination in order to avoid unsound results or a poor level of code reduction in slices with respect to the initial program.

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 is still capable of producing small slices, even in the 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. The implementation of these results in the Coq proof assistant is presented and some of its difficult points are discussed.

Keywords: Program slicing, Trajectory-based semantics, Verification, Run-time errors, Non-terminating loops, Coq formalization

1. Introduction

Context. Program slicing was initially introduced by Weiser [Wei81, Wei82] as a technique allowing to transform 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 the *slicing criterion*. A common slicing criterion is a program point

Correspondence and offprint requests to: J.-C. Léchenet, e-mail: jean-christophe.lechenet@cea.fr

l. 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. [Amt08, BH93, BBD+10, BDG+06, CF89, DBH+11, HRB88, RAB+07, RY89, RY88]). 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 [BH04, Sil12, Tip95, XQZ+05]. 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" [CKGJ12, GTXT11, HD95, KKPP15]. Our main goal is to address the following research question:

(RQ) How 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.

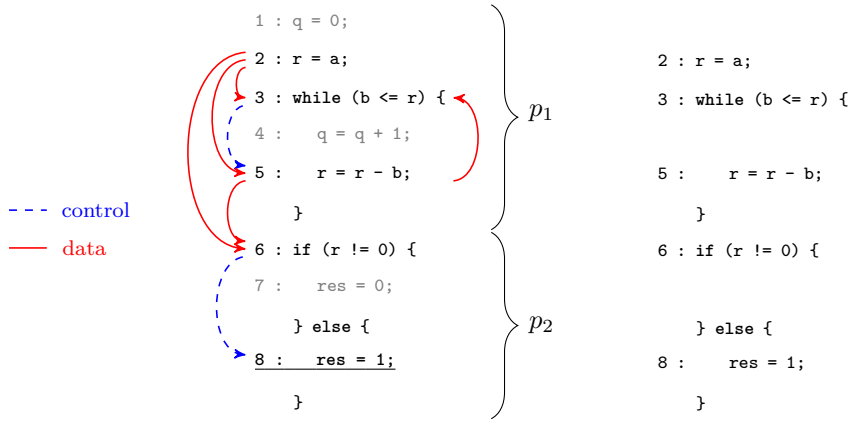
While some aspects of **(RQ)** were discussed in previous papers, none of them provided a complete formal answer in the considered general setting (as we argue in Sec. 3 and 8 below). To satisfy the traditional soundness property, program slicing would require to consider additional dependencies of each statement on previous loops and error-prone statements. That would lead to larger slices, where we would *systematically preserve all potentially erroneous or non-terminating statements* executed before the slicing criterion. Such slices would have a very limited benefit for our purpose of performing V&V on slices instead of the initial program.

This work proposes *relaxed slicing,* a slicing technique where additional dependencies on previously executed (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.

Relaxed slicing, its soundness property, and the justification of its use in V&V have been formalized in the Coq proof assistant [BC04] for a language representative for our purpose. The formalization is available in [Léc16]. One key difficulty of this formalization is related to data dependence whose definition does not follow the structure of the program. It significantly complicates proofs by induction. Our solution proposes to reformulate the dependence relations in an executable form suitable for the computation of a slice. We prove that it is equivalent to the original definitions. We present the reformulations and some aspects of the Coq development. A certified implementation of relaxed slicing is automatically extracted from the Coq formalization.

The contributions of this work include:

- a comprehensive analysis of issues arising for V&V on classic slices;
- the notion of relaxed slicing (Def. 5.6) for structured programs with possible errors and non-termination,



(a) Original program (b) Slice w.r.t. line 8

Fig. 1. A simple program deciding if `b` divides `a` (for `b` > 0 and `a` $\geqslant 0$) and its slice w.r.t. line 8

which keeps fewer statements than it would be necessary to satisfy the classic soundness property of slicing;

- a new soundness property for relaxed slicing (Th. 5.1);
- 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. 6.1, 6.2);
- a reformulation of the dependence relations, a description of the formalization and proof of our results in Coq;
- a certified implementation of relaxed slicing for the considered language extracted from the Coq formalization.

Paper outline. Sec. 2 presents the principles of static backward slicing and an example of its use in a combined V&V technique. Sec. 3 gives our motivation and illustrating examples. The considered language and its semantics are defined in Sec. 4. Sec. 5 defines the notion of relaxed slice and establishes its main soundness property. Next, Sec. 6 formalizes the relationship between the errors in the initial program and in a relaxed slice, while Sec. 7 presents the reformulations of the dependence relations, proves their correctness, and details the formalization of relaxed slicing in Coq. Finally, Sec. 8 and 9 present the related work and the conclusion with some future work. This paper is an extended version of [LKG16] which was mainly enriched by a presentation of the Coq development and its difficulties (in Sec. 7).

2. Static Backward Slicing and Verification

2.1. Static Backward Slicing

Let us illustrate the principles and properties of classic static backward slicing on a simple example presented in Fig. 1. The original program, in Fig. 1a, tests if `b` divides `a`, assuming that `b` > 0 and `a` $\geqslant 0$. It can be divided into two parts. The first one (p_1, lines 1–5) performs the Euclidean division of `a` by `b`. The second part (p_2, lines 6–8) tests the value of the remainder. At the end of the first part of the program, the quotient is stored in `q` and the remainder is stored in `r`. The quotient and the remainder are computed together by the program, but the quotient is actually not needed to get the final result and check if `b` divides `a`. Suppose we are interested in particular in the case where `b` divides `a`. In this case, the program assigns `res` to `1` at line 8. Suppose we want to remove from the program the statements that do not impact line 8. This is the purpose of static backward slicing [Wei81, BBD+10].

To compute the slice, we first need to select a slicing criterion. We select line 8, underlined in Fig. 1a, as our slicing criterion. We give below the soundness property establishing the link between the semantics

of the original program and its slice, but intuitively we want to obtain a reduced program that behaves in the same way as the original program with respect to line 8. For that, slicing should preserve the statements that, directly or indirectly, have an impact on the slicing criterion. This is expressed using control and data dependencies that we define informally hereafter, and formally in Sec. 5.1.

A statement s is *control-dependent* on another statement s' if s' can decide whether s will be executed. One classic source of control dependence is the use of conditional statements and loops. In our program, lines 7 and 8 depend on line 6, because they are in the branches of the condition on line 6. Likewise, lines 4 and 5 are control-dependent on line 3, because they are in the loop body of the `while` loop at line 3. The control dependencies involved in the computation of our slice are shown using blue dashed arrows in Fig. 1a.

A statement s is *data-dependent* on another statement s' if s' assigns a variable used by s and there exists a symbolic path from s' to s such that this variable is not reassigned meanwhile on this path. This corresponds to classic def-use paths computed using a reaching definition dataflow analysis. Data dependence captures the statements that impact the values of the variables appearing in the slicing criterion. In our example, line 6 depends both on lines 2 and 5, since `r` is used at line 6 and can be last assigned at line 2 (if the execution does not enter the loop body at lines 4–5) or at line 5. Other data dependencies due to variable `r` include the dependencies of line 5 on itself and on line 2, and those of line 3 on lines 2 and 5. Because of variable `q`, line 4 depends on itself and on line 1. Notice that the dependency of a statement on itself cannot lead to keeping any additional statement in the slice and can thus be ignored from our point of view. The data dependencies involved in the computation of our slice are shown using red solid arrows in Fig. 1a.

The slice is obtained by keeping the statements on which the slicing criterion is, directly or indirectly, control-dependent or data-dependent. In Fig. 1a, the arrows indicate the control and data dependencies that are relevant with respect to the slicing criterion and involved in the computation of the slice. The statements that are not preserved are lines 1, 4 and 7. They are grayed out in Fig. 1a. The resulting slice is represented in Fig. 1b.

Classic soundness property. Most previous applications of slicing to debugging use slices in order to *better understand an already detected error,* by analyzing a simpler program rather than a more complex one [BDG+06, Sil12, Tip95]. 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 the absence or presence of errors in a slice in terms of the initial program requires solid theoretical foundations. The link between the original program and its slice is made by the soundness property relating their semantics. The classic soundness property of slicing (cf. [BBD+10, Def. 2.5] or [RY88, Slicing Th.]) can be informally stated as follows.

Property 2.1. *Let p be a program, q a slice of p w.r.t. a slicing criterion l, and 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.*[1]

This property actually states that the program and its slice have the same behavior not only with respect to the slicing criterion, but also with respect to all statements preserved in the slice. For this reason, it appears desirable to apply slicing in order to detect certain behaviors of the original program on the slices rather than on the original program. One example is the detection of errors during V&V. The benefit of slicing would be related to the size of the analyzed programs. Indeed, the slices are expected to be smaller than the original program, so the analysis of slices can become less costly than for the original program.

But can the results obtained on the slices be transposed to the initial program directly? In other words, we should answer the research questions **(RQ)** presented in Sec. 1. If no error is found in a slice, can we deduce that no error occurs in the initial program, at least inside the statements in common with the slice? And conversely, if an error is found, can we deduce that the same error occurs in the initial program?

Unfortunately, the classic soundness property cannot be used in this context. Indeed, it 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. While recent research efforts tackle this problem (cf. Sec. 8), none of them provide a complete formalization in the general case with errors and non-termination. We will present in Sec. 3 an example illustrating why this property does not hold in the presence of potential runtime errors or non-terminating loops. Let us illustrate now how verification methods can take advantage of program slicing.

[1] Formally, using the notation introduced hereafter in the paper (cf. Def. 5.8), their *projections* are equal: $\mathsf{Proj}_L(\mathcal{T}[\![p]\!]\sigma) = \mathsf{Proj}_L(\mathcal{T}[\![q]\!]\sigma)$.

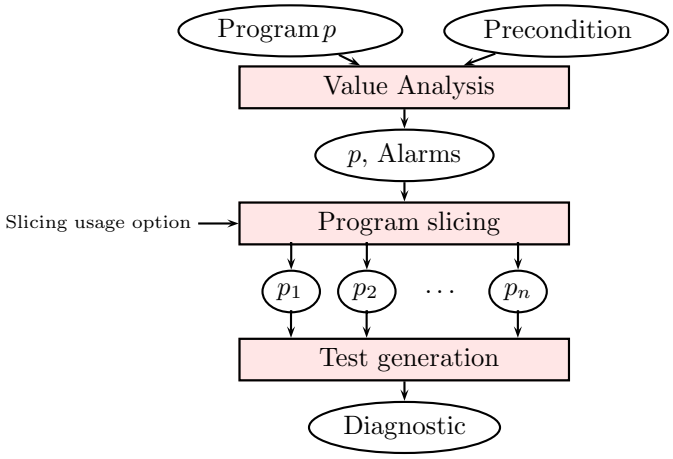


Fig. 2. Overview of the SANTE method

2.2. The SANTE Method

One method applying verification on slices is the SANTE method [CKGJ11, CKGJ12, CCK+14] that aims at detecting runtime errors in a given C program. It combines static analysis, program slicing and test generation as illustrated by Fig. 2. It takes as inputs a C program p and a precondition which defines acceptable input values of p (e.g. value ranges for input variables of p and required relationships between them).

The first step applies abstract interpretation based value analysis to produce a set of *alarms* A reporting threatening statements in p for which it detects a risk of runtime error. This includes in particular the risks of division by zero, out-of-bounds array access and some cases of invalid pointers.

The objective of the second step, based on program slicing, is to simplify the initial program before the last step. According to the user-defined slicing option and the structure of dependencies in A, this step determines which and how many slices should be generated and sent to dynamic analysis. Each slice contains a subset of alarms that can be triggered.

Finally, for each slice p_i, the dynamic analysis step tries to activate the potential threat for each alarm present in p_i. It is based on all-path test generation that may generate a *counter-example* for an alarm, i.e. a test datum showing that the threatening statement detected by value analysis as an alarm is not safe and really provokes a runtime error. It allows SANTE to produce for each alarm a *diagnostic* that can be *safe* for a false alarm (when all paths are explored without triggering an error), *bug* for an effective bug (confirmed by some input state), or *unknown* if it cannot be decided whether this alarm is an effective error or not. An alarm is said to be *classified* if its diagnostic is bug or safe.

The SANTE tool relied on an implementation of slicing that was basically equivalent to the relaxed slicing formally introduced in the present work. The soundness of this approach was understood and conjectured by the authors of SANTE but was not formally proved.

First experiments with SANTE [CKGJ12, CCK+14] show that relaxed slicing allows the tool to reduce the program on average by 51% (going up to 97% for some examples) before the dynamic analysis step, and accelerates the V&V task on average by 43% compared to the approach in which the dynamic analysis is run on the original program without slicing. Since the dynamic analysis is run on shorter programs, SANTE exhibits counter-examples with shorter paths (that make it easier for the verification engineer to identify the error). The path length in counter-examples is 24% smaller on average, and this reduction rate goes up to 71% on some programs. In terms of results, the number of unclassified (*unknown*) alarms is decreased on average by 82% in the variant of the method that creates one slice per alarm. The SANTE method was illustrated in detail on an example program called `eurocheck` in [CKGJ11].

As shown above, SANTE advantageously combines slicing with other verification approaches and uses slices to decide whether some errors can occur in the initial program. But it lacks a solid theoretical foundation to provide a high level of trust in the obtained results. Such a theoretical foundation is provided by this paper.

```
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. 3. (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.

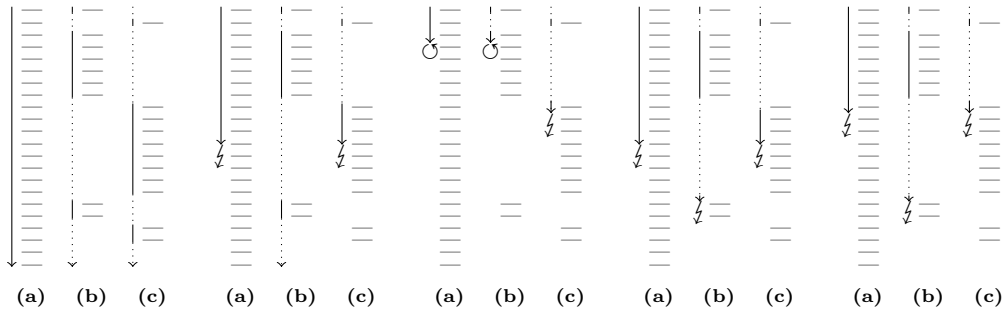
Initial state		σ_1	σ_2	σ_3	σ_4	σ_5
Inputs		$k = 2$ $N = 5$	$k = 2$ $N = 4$	$k = 0$ $N = 4$	$k = 2$ $N = 0$	$k = 0$ $N = 0$
Example array		`{3,0,4,0,3}`	`{3,0,1,0}`	`{12,0,0,0}`	`{}`	`{}`
Anomaly	**(a)**	—	↯ line 13	↺ line 4	↯ line 13	↯ line 10
	(b)	—	—	↺ line 4	↯ line 17	↯ line 17
	(c)	—	↯ line 13	↯ line 10	↯ line 13	↯ line 10
Schematic behavior						

Fig. 4. Errors (↯), non-termination (↺) and normal termination (—) of programs of Fig. 3 for some inputs.

3. Motivation and Running Examples

Errors and assertions. First of all, let us specify what kind of errors we consider and how we model them. We consider errors that are determined by the current program state[2] including runtime errors (division by

[2] Temporal errors (e.g. use-after-free in C) cannot be directly represented in this way.

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. A similar assumption is made in the SANTE method. This assumption will be convenient for the formalization in the next sections: possible runtime errors will only occur in assertions. Such assertions can be generated syntactically (for example, by the RTE plugin of the FRAMA-C toolset [KKP+15] for C programs). For instance, line 10 in Fig. 3a 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 also be used to express any additional user-defined properties on the current state.

Illustrating examples. Let us illustrate on a few simple examples that the relationship between the presence and the absence of errors in the original program and in the slice is not so obvious as it might appear at first glance. Fig. 3a presents a simple (buggy) C-like program that takes as inputs an array `a` of length `N` and an integer `k` (with $0 \leqslant \texttt{k} \leqslant 100$, $0 \leqslant \texttt{N} \leqslant 100$), and computes in two different ways the average of the elements of `a`. Thus, the valid indices of array `a` range between 0 and `N-1`. 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 multiple 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. 3b 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.[3] In addition, we keep the assertions to prevent potential errors in preserved statements. Similarly, Fig. 3c shows a slice with respect to line 20, again with protecting assertions.

Fig. 4 summarizes the behavior of the three programs of Fig. 3 on some test data. The values of the elements of `a` do not matter here, but concrete examples of input are given nevertheless to help understand the example. The last line illustrates the behavior of programs **(a)**, **(b)** and **(c)** on the test data. More precisely, we use two icons, $\circlearrowleft$ and $\lightning$, to represent respectively a non-terminating loop and an error. Normal executions are represented by a full-length arrow. For each test datum, the behavior of the three programs is schematically described using these graphic conventions in order to facilitate the comparison between them.

Let us use the test datum σ_1 to illustrate the behavior of the program. As assumed above, only those elements of `a` whose indices are multiple of $\texttt{k} = 2$ can be nonzero. Thus, instead of computing the sum of the elements of `a` as `a[0] + a[1] + a[2] + a[3] + a[4]`, we can compute it as `a[0] + a[2] + a[4]`. Then the average of `a` can be computed by dividing this sum by the length of `a` (here, $\texttt{N} = 5$). The program of Fig. 3a computes this simplified sum in two different ways. The first loop (lines 4–8) increments the index `i` by `k` while `i` is less than `N`. This can lead to an infinite loop if $\texttt{k} = 0$ (cf. σ_3). Line 11 tries to compute the number of multiples of `k` to be considered in the second loop as `N/k` (which obviously fails if $\texttt{k} = 0$, cf. σ_5). The indices used in the second loop are thus the multiples of `k` ranging between `k * 0` and `k * (N/k)`. For σ_1, $\texttt{N/k} = 5/2 = 2$, and the indices of `a` considered are the right ones: 0, 2 and 4. But this is not the right number of iterations if `k` divides `N`. Indeed, in this case, the last index considered by the second loop is equal to `N` that is invalid (cf. σ_2). Finally, the average of `a` is computed on lines 18 and 20 by dividing the sums by `N`. Here again, these divisions fail if $\texttt{N} = 0$ (cf. slice **(b)** on input σ_4).

Executing slice **(b)** on σ_2 does not reveal any error, whereas executing program **(a)** on the same input gives an error at line 13. Line 13 is not preserved in **(b)**. This shows that a slice and the original program can diverge due to an error triggered by a non-preserved statement. Conversely, consider the 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

[3] By formal definitions of Sec. 5, 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.

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, while 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 2.1 is not true in case of the 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 is not preserved in the slice, or
(ii) a previously executed failing statement is 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. 3, 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 [HSD96, PC90, RAB+07], cf. Sec. 8) proposes to ensure Property 2.1 even in the 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.

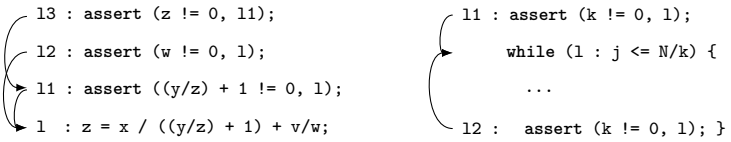
The formalization of relaxed slicing is presented in the following manner. First, we provide a paper-and-pencil formalization of the considered language, dependence relations, relaxed slicing and its properties in Sec. 4–6. Next we focus on the Coq formalization. Sec. 7.1 presents alternative definitions of dependence relations as functions that appeared to be necessary to perform the formalization in Coq. Finally, we discuss the implementation in Coq and emphasize some of its difficult points in Sec. 7.2.

4. 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 the presence of runtime errors and non-termination. The language is defined by the following grammar:

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

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 intoduce the distinction between Prog and Stmt to avoid the problems related to the associativity of ";". Defining programs as lists



(a) Chained assertions (b) Loop condition

Fig. 5. Two special cases of assertions

of statements allows us to consider empty programs and to manipulate statements without being hampered by associativity of ";" for statements. 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 **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. 3, where the protected label is not indicated). Two simple cases however need more flexibility (cf. Fig. 5). Some assertions have to be themselves protected by assertions when they contain a threatening expression. Fig. 5a 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. 5b (inspired by the second loop of Fig. 3) shows how assertions with explicit labels can be used to protect a loop condition from a runtime error. The arrows in Fig. 5 indicate the protected statement.

Assertions can also be 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 also be 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 the error state. We consider only valid initial states. Let $\sigma \in \Sigma$ 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 $\langle (l_1, \sigma_1) \dots (l_k, \sigma_k) \dots \rangle$, where $l_1, \dots, l_k, \dots$ is the sequence of labels of the executed instructions, and $\sigma_i \in \Sigma_\varepsilon$ 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 and a state $\sigma \in \Sigma$, let $LS_\sigma(T)$ be 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 only occur in assertions. We also suppose for simplicity that all variables appearing in p are initialized in any initial state $\sigma \in \Sigma$ of p, which 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. 6 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. 6. For any state $\sigma \in \Sigma$, 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

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

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

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

Fig. 6. Trajectory-based semantics of the language (for a valid state $\sigma \in \Sigma$)

ends with the error state. Indeed, 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. 3, 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 the trajectory on σ_3 is infinite, while the trajectory on σ_5 leads to an error:

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

5. Relaxed Program Slicing

5.1. Control and Data Dependences

Let $L(p)$ denote the set of labels of program p. From now on, we consider 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 (like in some strategies of the SANTE method [CKGJ12]). 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 $\mathcal{D}$ in p, i.e. $(l, l') \in \mathcal{D}$. In the remainder of this paper, as there will be no ambiguity about the program concerned, we will omit p and use the simpler notation $l \xrightarrow{\mathcal{D}} l'$. The definitions of control and data dependencies, denoted respectively $\mathcal{D}_c$ and $\mathcal{D}_d$, are standard, and given following [BBD+10].

Definition 5.1 (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* $l \xrightarrow{\mathcal{D}_c} l'$;
for any statement $\texttt{while}\ (l : b)\ q$ *and* $l' \in L(q)$, *we define* $l \xrightarrow{\mathcal{D}_c} l'$.

For instance, in Fig. 3a, 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 5.2 (Finite syntactic paths)**.** *The set of finite syntactic paths $\mathcal{P}(p)$ of a program p is inductively defined as follows:*

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

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

Definition 5.3 (Data dependence $\mathcal{D}_d$)**.** *Let l and l' be labels of a program p. We say that there is a data dependency $l \xrightarrow{\mathcal{D}_d} l'$ if $\mathsf{def}(l) \neq \varnothing$ and $\mathsf{def}(l) \subseteq \mathsf{ref}(l')$ and there exists a path $\pi = \pi_1 l \pi_2 l' \pi_3 \in \mathcal{P}(p)$ such that for all $l'' \in \pi_2$, $\mathsf{def}(l'') \neq \mathsf{def}(l)$. Each π_i may be empty.*

For instance, in Fig. 3b, 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 not all subsets of the labels of a program define a quotient. Indeed, 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. Therefore, a set of labels containing a label inside a branch (or a loop body) without the label of the corresponding conditional (or loop) statement cannot be the set of labels of a quotient.

The slice will be defined below as the set of labels of preserved statements. To make it well-defined, we need to show first that such a set of labels is the set of labels of a quotient. This is the purpose of Lemma 5.1, which shows that this property holds as long as the considered dependence relation includes control dependence, which is always the case in practice.

Given a dependence relation $\mathcal{D}$ and L_0 a set of labels, 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' such that there exists $l \in L_0$ with $l' \xrightarrow{\mathcal{D}^*} l$.

Lemma 5.1. *Let p be a program, L_0 a subset of labels of p and $\mathcal{D}$ a dependence relation on p satisfying $\mathcal{D}_c \subseteq \mathcal{D}$. Then $(\mathcal{D}^*)^{-1}(L_0)$ is the set of labels of a (uniquely defined) quotient of p.*

Proof. We note $L = (\mathcal{D}^*)^{-1}(L_0)$. Notice that $L \subseteq L(p)$. To show the existence of a quotient of p whose set of labels is L, we construct such a quotient explicitly.

We define inductively the function $\mathcal{F}_L$ over *Prog* as follows.

- For an empty program:

  $\mathcal{F}_L(\lambda) = \lambda$

- For a sequence of statements:

  $\mathcal{F}_L(s;\ p) = \mathcal{F}_L(s); \mathcal{F}_L(p)$

- For a **skip** statement, an assignment or an assertion s with label l:

  $$\mathcal{F}_L(s) = \begin{cases} s & \text{if } l \in L \\ \lambda & \text{otherwise} \end{cases}$$

- For an **if** statement:

  $$\mathcal{F}_L(\texttt{if } (l:b)\ q \texttt{ else } r) = \begin{cases} \texttt{if } (l:b)\ \mathcal{F}_L(q) \texttt{ else } \mathcal{F}_L(r) & \text{if } l \in L \\ \lambda & \text{otherwise} \end{cases}$$

- For **while** statements:

  $$\mathcal{F}_L(\texttt{while } (l:b)\ q) = \begin{cases} \texttt{while } (l:b)\ \mathcal{F}_L(q) & \text{if } l \in L \\ \lambda & \text{otherwise} \end{cases}$$

To prove that $\mathcal{F}_L(p)$ is a quotient of p containing exactly the labels of L, we first claim that:

1. for any program q, $\mathcal{F}_L(q)$ is a quotient of q;
2. for any quotient q of p, the set of labels of $\mathcal{F}_L(q)$ is $L \cap L(q)$.

Claim 1 holds for any arbitrary program q, whereas Claim 2 requires the hypothesis on control dependence ($\mathcal{D}_c \subseteq \mathcal{D}$), which explains why it focuses only on quotients of p. Both claims are proved by induction on q. We detail below the proof of the claims for the case of the conditional statement (the complete proof being available in the Coq development).

To prove Claim 1, assume that both $\mathcal{F}_L(q)$ and $\mathcal{F}_L(r)$ are quotients of q and r respectively and let us prove that $\mathcal{F}_L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r)$ is a quotient of $\texttt{if}\ (l : b)\ q\ \texttt{else}\ r$. If $l \notin L$, we have $\mathcal{F}_L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r) = \lambda$ which is a quotient of every program and thus a quotient of $\texttt{if}\ (l : b)\ q\ \texttt{else}\ r$. If $l \in L$, $\mathcal{F}_L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r) = \texttt{if}\ (l : b)\ \mathcal{F}_L(q)\ \texttt{else}\ \mathcal{F}_L(r)$. The statements removed by $\mathcal{F}_L$ from q and r can be removed directly from $\texttt{if}\ (l : b)\ q\ \texttt{else}\ r$ to construct $\texttt{if}\ (l : b)\ \mathcal{F}_L(q)\ \texttt{else}\ \mathcal{F}_L(r)$ which is therefore a quotient of $\texttt{if}\ (l : b)\ q\ \texttt{else}\ r$ by definition.
To prove Claim 2, assume that $L(\mathcal{F}_L(q)) = L(q) \cap L$ and $L(\mathcal{F}_L(r)) = L(r) \cap L$ and let us prove that $L(\mathcal{F}_L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r)) = L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r) \cap L$. By definition, $L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r) \cap L = (\{l\} \cup L(q) \cup L(r)) \cap L = (\{l\} \cap L) \cup (L(q) \cap L) \cup (L(r) \cap L)$. Assume first that $l \notin L$. Then we have $L(\mathcal{F}_L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r) = L(\lambda) = \varnothing$. Since $l \notin L$, thanks to the hypothesis on control dependence, we can deduce that $L(q) \cap L = \varnothing$ and $L(r) \cap L = \varnothing$. Therefore, $(\{l\} \cap L) \cup (L(q) \cap L) \cup (L(r) \cap L) = \varnothing \cup \varnothing \cup \varnothing = \varnothing$. Thus, for the case $l \notin L$, we can conclude that $L(\mathcal{F}_L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r)) = L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r) \cap L$. Assume now that $l \in L$. In this case, $\mathcal{F}_L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r) = \texttt{if}\ (l : b)\ \mathcal{F}_L(q)\ \texttt{else}\ \mathcal{F}_L(r)$. Thus, we have $L(\mathcal{F}_L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r)) = \{l\} \cup L(\mathcal{F}_L(q)) \cup L(\mathcal{F}_L(r)) = \{l\} \cup (L(q) \cap L) \cup (L(r) \cap L)$. And since $l \in L$, we have $L(\mathcal{F}_L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r)) = (\{l\} \cap L) \cup (L(q) \cap L) \cup (L(r) \cap L)$, which gives $L(\mathcal{F}_L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r)) = L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r) \cap L$. This concludes the proof of the claims for the conditional case.

The end of the proof is straightforward. By Claim 1, $\mathcal{F}_L(p)$ is a quotient of p. It follows then from Claim 2 that $L(\mathcal{F}_L(p)) = L(p) \cap L = L$. Therefore $\mathcal{F}_L(p)$ is a quotient of p containing exactly the labels of $L = (\mathcal{D}^*)^{-1}(L_0)$. □

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 5.4 (Dependence-based slice)**.** *Let $\mathcal{D}$ be a dependence relation on p such that $\mathcal{D}_c \subseteq \mathcal{D}$, and L_0 a set of labels. 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 $\mathcal{D} = \mathcal{D}_c \cup \mathcal{D}_d$.*

5.2. Assertion Dependence and Relaxed Slices

Soundness of classic slicing for programs without runtime errors or non-terminating loops can be expressed by Property 2.1 in Sec. 2. As we illustrated, to generalize this property in the 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 5.5 (Assertion dependence $\mathcal{D}_a$)**.** *For every assertion l : $\texttt{assert}\ (b, l')$ in p with $l, l' \in L(p)$, we define an assertion dependency $l \xrightarrow{\mathcal{D}_a} l'$.*

To illustrate the definition, we can use Fig. 5 again. The arrows illustrate the protection of a statement by an assertion, and thus correspond to assertion dependence.

Definition 5.6 (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. 3a, 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. 3

(in which we artificially preserved assertions in Sec. 3) are in fact relaxed slices where assertions are naturally preserved thanks to the assertion dependence.

Assertion dependence brings two benefits. First, it ensures that a potentially threatening instruction is never kept without its protecting assertion. Second, an assertion can be preserved without its protected statement, which is quite useful for V&V that focus on assertions. Indeed, 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 of the program in Fig. 3a 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 [HSD96, PC90, RAB+07]). For example, slice **(b)** does not include the potential error at line 13, and slice **(c)** does not include the loop of line 4.

5.3. 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 5.7 (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 5.8 (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:*

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

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

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

Theorem 5.1 (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:*

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

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

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

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. 5.8, the projections of $\mathcal{T}[\![q]\!]\sigma$ and T to $L = L(q)$ have the following form

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

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 $(\mathsf{Proj}_L(T))^k = \mathsf{Proj}_L((\mathcal{T}[\![q]\!]\sigma)^k)$, 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. 5.8 we have

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

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

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 $\sigma'_k{\downarrow}\,\mathsf{used}(l'_k) = \sigma_{f(k)}{\downarrow}\,\mathsf{used}(l_{f(k)})$. 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. 5.1). 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 $\sigma'_k{\downarrow}\,\mathsf{ref}(l'_{k+1}) = \sigma_{f(k+1)-1}{\downarrow}\,\mathsf{ref}(l_{f(k+1)})$. Indeed, assume that there is a variable $v \in \mathsf{ref}(l'_{k+1}) = \mathsf{ref}(l_{f(k+1)})$ 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. 5.3), 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: $\sigma'_u{\downarrow}\,\mathsf{used}(l'_u) = \sigma_{f(u)}{\downarrow}\,\mathsf{used}(l_{f(u)})$, 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 $\mathsf{Proj}_L(T) = \mathsf{Proj}_L(T')$.

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 $\mathsf{Proj}_L(\mathcal{T}[\![p]\!]\sigma) = \mathsf{Proj}_L(T')$. 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, $T' = \mathcal{T}[\![q]\!]\sigma$. □

6. 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 6.1. *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. 5.1 applied to $T = \mathcal{T}[\![p]\!]\sigma$, it follows that there exists a finite prefix T' of $\mathcal{T}[\![q]\!]\sigma$ such that $\mathsf{Proj}_L(T) = \mathsf{Proj}_L(T')$. The last state of $\mathsf{Proj}_L(T')$ 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. 4) 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 6.1.

Theorem 6.1. *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 6.1. *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 6.2. *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.*

$$
\begin{array}{ll}
\mathsf{assert}\,(\lambda) & = \varnothing \\
\mathsf{assert}\,(s; p) & = \mathsf{assert}\,(s) \cup \mathsf{assert}\,(p) \\
\mathsf{assert}\,(l : \texttt{skip}) & = \varnothing \\
\mathsf{assert}\,(l : x = e) & = \varnothing \\
\mathsf{assert}\,(\texttt{if}\ (l : b)\ p\ \texttt{else}\ q) & = \mathsf{assert}\,(p) \cup \mathsf{assert}\,(q) \\
\mathsf{assert}\,(\texttt{while}\ (l : b)\ p) & = \mathsf{assert}\,(p) \\
\mathsf{assert}\,(l : \texttt{assert}\ (b, l')) & = \{(l, l')\}
\end{array}
$$

Fig. 7. Definition of assert, which computes assertion dependence

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 6.1 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. 5.1 can be applied and thus $\mathsf{Proj}_L(\mathcal{T}[\![p]\!]\sigma) = \mathsf{Proj}_L(\mathcal{T}[\![q]\!]\sigma)$. 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. 3 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. 6.1 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. 3, we can show that neither of the two slices of Fig. 3 contains any error. Since these slices cover all assertions, we can deduce by Cor. 6.1 that the initial program is error-free.

Th. 6.2 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. 6.2 can be established for non-termination: if $\mathcal{T}[\![q]\!]\sigma$ is infinite, then either (††) or (†††) holds for p.

In the framework of a verification method like SANTE, Cor. 6.1 and Th. 6.2 provide precise answers to the research question **(RQ)** and indicate how to safely transpose to the initial program the verification results obtained on the slices.

7. Coq Development

The structure of the Coq development [Léc16] basically follows the formalization of this paper. We describe the main steps and some difficult points. First, we present the reformulated definitions of dependence relations that we used in the Coq formalization and that appeared to be particularly helpful in the case of data dependence. Next, we describe the language, main steps of the development and soundness results.

7.1. Reformulation of the Three Dependence Relations

The definitions of the three dependence relations given in Sec. 5.1 cannot be used directly in an algorithm. Before presenting the Coq formalization, we introduce alternative definitions of dependence, that we will prove equivalent to the former ones, but that will be adapted to computation.

Let us begin with the easiest function: the one computing assertion dependence, implementing Def. 5.5. It is straightforward to write, since it only has to collect the labels from the `assert` statements. The function $\mathsf{assert} : \mathit{Prog} \to 2^{L \times L}$ is given in Fig. 7.

For this language, the function computing control dependence is also easy to write. It collects the labels inside the branches of the conditions and the bodies of the loops. The difference with Def. 5.1 is that

$$
\begin{aligned}
&\mathsf{top\_labels}\,(\lambda) && = \varnothing \\
&\mathsf{top\_labels}\,(s; p) && = \mathsf{top\_labels}\,(s) \cup \mathsf{top\_labels}\,(p) \\
&\mathsf{top\_labels}\,(l : \texttt{skip}) && = \{l\} \\
&\mathsf{top\_labels}\,(l : x = e) && = \{l\} \\
&\mathsf{top\_labels}\,(\texttt{if}\ (l : b)\ p\ \texttt{else}\ q) && = \{l\} \\
&\mathsf{top\_labels}\,(\texttt{while}\ (l : b)\ p) && = \{l\} \\
&\mathsf{top\_labels}\,(l : \texttt{assert}\ (b, l')) && = \{l\}
\end{aligned}
$$

Fig. 8. Definition of top_labels, which computes the set of labels of the top-level statements

$$
\begin{aligned}
&\mathsf{control}\,(\lambda) && = \varnothing \\
&\mathsf{control}\,(s; p) && = \mathsf{control}\,(s) \cup \mathsf{control}\,(p) \\
&\mathsf{control}\,(l : \texttt{skip}) && = \varnothing \\
&\mathsf{control}\,(l : x = e) && = \varnothing \\
&\mathsf{control}\,(\texttt{if}\ (l : b)\ p\ \texttt{else}\ q) && = \{(l, l') \mid l' \in \mathsf{top\_labels}\,(p) \cup \mathsf{top\_labels}\,(q)\} \cup \mathsf{control}\,(p) \cup \mathsf{control}\,(q) \\
&\mathsf{control}\,(\texttt{while}\ (l : b)\ p) && = \{(l, l') \mid l' \in \mathsf{top\_labels}\,(p)\} \cup \mathsf{control}\,(p) \\
&\mathsf{control}\,(l : \texttt{assert}\,(b, l')) && = \varnothing
\end{aligned}
$$

Fig. 9. Definition of control, which computes control dependence

this function captures unitary (i.e. direct) dependencies only. It associates the label of a condition (resp. a loop) with the labels of the top-level statements in its branches (resp. its body), and does not enter nested conditions or loops. As the slice is computed using reflexive and transitive closures, this does not change the final result. The function $\mathsf{top\_labels} : Prog \to L$ computing the labels of the top-level statements of a program is presented in Fig. 8. It is used to define the function $\mathsf{control} : Prog \to 2^{L \times L}$ (shown in Fig. 9) implementing the computation of control dependence of Def. 5.1.

Contrary to the two previous functions, the computation of data dependence is not immediate. Indeed, a difficulty arises as soon as the program contains some loop, since the definition allows to unroll loops any (unbounded) number of times to find new data dependencies. Moreover, whereas all other definitions are expressed in a recursive way along the structure of the programs, the definition of data dependence of Def. 5.3 is not. It is illustrated in Fig. 10, where, considering a sequence of two programs p_1 and p_2, data dependencies can come from programs p_1 and p_2, but also from def-use paths starting in p_1 and ending in p_2.

To manipulate only recursive definitions, we introduce three functions that can be computed recursively: mustdef, maydef and mayref.

Definition 7.1 (Auxiliary sets mustdef, maydef, mayref)**.** *Let p be a program, Id the set of identifiers in our language, and let* $\mathsf{vars}\,(e)$ *denote the set of identifiers occurring in expression e.*

- *The function* $\mathsf{mustdef} : Prog \to 2^{Id}$ *is defined as shown in Fig. 11.*
- *The function* $\mathsf{maydef} : Prog \to 2^{Id \times L}$ *is defined as shown in Fig. 12.*
- *The function* $\mathsf{mayref} : Prog \to 2^{Id \times L}$ *is defined as shown in Fig. 13.*

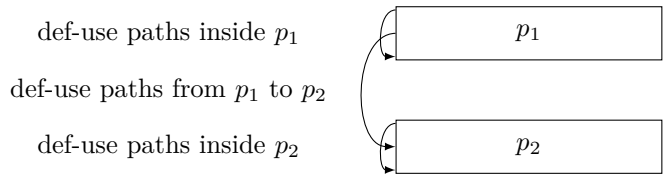


Fig. 10. Possible data dependencies for a sequence of programs $p_1; p_2$

$$
\begin{aligned}
&\mathsf{mustdef}\ (\lambda) && = \varnothing \\
&\mathsf{mustdef}\ (s; p) && = \mathsf{mustdef}\ (s) \cup \mathsf{mustdef}\ (p) \\
&\mathsf{mustdef}\ (l : \texttt{skip}) && = \varnothing \\
&\mathsf{mustdef}\ (l : x = e) && = \{x\} \\
&\mathsf{mustdef}\ (\texttt{if}\ (l : b)\ p\ \texttt{else}\ q) && = \mathsf{mustdef}\ (p) \cap \mathsf{mustdef}\ (q) \\
&\mathsf{mustdef}\ (\texttt{while}\ (l : b)\ p) && = \varnothing \\
&\mathsf{mustdef}\ (l : \texttt{assert}\ (b, l')) && = \varnothing
\end{aligned}
$$

Fig. 11. Definition of mustdef

$$
\begin{aligned}
&\mathsf{maydef}\ (\lambda) && = \varnothing \\
&\mathsf{maydef}\ (s; p) && = \{(x, l) \mid (x, l) \in \mathsf{maydef}\ (s) \wedge x \notin \mathsf{mustdef}\ (p)\} \cup \mathsf{maydef}\ (p) \\
&\mathsf{maydef}\ (l : \texttt{skip}) && = \varnothing \\
&\mathsf{maydef}\ (l : x = e) && = \{(x, l)\} \\
&\mathsf{maydef}\ (\texttt{if}\ (l : b)\ p\ \texttt{else}\ q) && = \mathsf{maydef}\ (p) \cup \mathsf{maydef}\ (q) \\
&\mathsf{maydef}\ (\texttt{while}\ (l : b)\ p) && = \mathsf{maydef}\ (p) \\
&\mathsf{maydef}\ (l : \texttt{assert}\ (b, l')) && = \varnothing
\end{aligned}
$$

Fig. 12. Definition of maydef

Intuitively, $\mathsf{mustdef}\ (p)$ is the set of variables surely defined (i.e. assigned) in p; $\mathsf{maydef}\ (p)$ is the set of pairs (x, l) where l is the label of one of possible last assignments to x in p; and $\mathsf{mayref}\ (p)$ is the set of pairs (x, l) where l is the label of one of possible statements reading x without x being previously assigned in p. This intuition is formalized by Lemma 7.1 stated below. Notice that maydef and mayref both return sets of pairs (variable, label), while mustdef only returns a set of variables.

We illustrate these definitions on the program of Fig. 1a. To illustrate Def. 7.1 in the case of the sequence, we reuse the separation into p_1 (lines 1–5) and p_2 (lines 6–8). By applying the definitions, we have:

$$
\begin{aligned}
&\mathsf{mustdef}\ (p_1) = \{\mathtt{q}, \mathtt{r}\}, \quad \mathsf{maydef}\ (p_1) = \{(\mathtt{q}, 1), (\mathtt{r}, 2), (\mathtt{q}, 4), (\mathtt{r}, 5)\}, \quad \mathsf{mayref}\ (p_1) = \{(\mathtt{a}, 2), (\mathtt{b}, 3), (\mathtt{b}, 5)\} \\
&\mathsf{mustdef}\ (p_2) = \{\texttt{res}\}, \quad \mathsf{maydef}\ (p_2) = \{(\texttt{res}, 7), (\texttt{res}, 8)\}, \qquad\qquad \mathsf{mayref}\ (p_2) = \{(\mathtt{r}, 6)\}
\end{aligned}
$$

Functions maydef and mayref keep track, respectively, of possible prefixes and suffixes of def-use paths. This is the meaning of the following result, whose proof can be found in the Coq development [Léc16].

Lemma 7.1 (Properties of mustdef, maydef, mayref)**.** *Let p be a program, x an identifier and l a label.*

(i) $x \notin \mathsf{mustdef}\ (p)$ *if and only if there exists a syntactic path π of p such that for all $l' \in \pi$, $x \notin \mathsf{def}(l')$.*

(ii) $(x, l) \in \mathsf{maydef}\ (p)$ *if and only if there exists a syntactic path $\pi_1 l \pi_2$ of p such that $x \in \mathsf{def}(l)$ and for all l' in π_2, $x \notin \mathsf{def}(l')$.*

(iii) $(x, l) \in \mathsf{mayref}\ (p)$ *if and only if there exists a syntactic path $\pi_1 l \pi_2$ of p such that $x \in \mathsf{ref}(l)$ and for all l' in π_1, $x \notin \mathsf{def}(l')$.*

$$
\begin{aligned}
&\mathsf{mayref}\ (\lambda) && = \varnothing \\
&\mathsf{mayref}\ (s; p) && = \mathsf{mayref}\ (s) \cup \{(x, l) \mid (x, l) \in \mathsf{mayref}\ (p) \wedge x \notin \mathsf{mustdef}\ (s)\} \\
&\mathsf{mayref}\ (l : \texttt{skip}) && = \varnothing \\
&\mathsf{mayref}\ (l : x = e) && = \{(y, l) \mid y \in \mathsf{vars}\ (e)\} \\
&\mathsf{mayref}\ (\texttt{if}\ (l : b)\ p\ \texttt{else}\ q) && = \{(x, l) \mid x \in \mathsf{vars}\ (b)\} \cup \mathsf{mayref}\ (p) \cup \mathsf{mayref}\ (q) \\
&\mathsf{mayref}\ (\texttt{while}\ (l : b)\ p) && = \{(x, l) \mid x \in \mathsf{vars}\ (b)\} \cup \mathsf{mayref}\ (p) \\
&\mathsf{mayref}\ (l : \texttt{assert}\ (b, l')) && = \{(x, l) \mid x \in \mathsf{vars}\ (b)\}
\end{aligned}
$$

Fig. 13. Definition of mayref

$$
\begin{aligned}
&\mathsf{data}\,(\lambda) && = \varnothing\\
&\mathsf{data}\,(s;p) && = \mathsf{data}\,(s) \cup \mathsf{data}\,(p) \cup \{(l_1,l_2) \mid \exists x, (x,l_1) \in \mathsf{maydef}\,(s) \wedge (x,l_2) \in \mathsf{mayref}\,(p)\}\\
&\mathsf{data}\,(l : \texttt{skip}) && = \varnothing\\
&\mathsf{data}\,(l : x = e) && = \varnothing\\
&\mathsf{data}\,(\texttt{if}\ (l:b)\ p\ \texttt{else}\ q) && = \mathsf{data}\,(p) \cup \mathsf{data}\,(q)\\
&\mathsf{data}\,(\texttt{while}\ (l:b)\ p) && = \mathsf{data}\,(p) \cup \{(l_1,l_2) \mid \exists x, (x,l_1) \in \mathsf{maydef}\,(p) \wedge (x,l_2) \in \mathsf{mayref}\,(\texttt{while}\ (l:b)\ p)\}\\
&\mathsf{data}\,(l : \texttt{assert}\ (b,l')) && = \varnothing
\end{aligned}
$$

Fig. 14. Definition of data

In our example, $(\mathtt{r}, 2) \in \mathsf{maydef}\,(p_1)$ where an associated syntactic path is 1, 2 (since $\mathtt{r} \notin \mathsf{def}(3)$). Likewise, $(\mathtt{r}, 5) \in \mathsf{maydef}\,(p_1)$ where one associated path is 1, 2, 3, 4, 5, 3 (since $\mathtt{r} \notin \mathsf{def}(3)$). We also have $(\mathtt{r}, 6) \in \mathsf{mayref}\,(p_2)$ with an associated path 6, 7, but $(\mathtt{r}, 5) \notin \mathsf{ref}(p_1)$ since every syntactic path begins with $1, 2$ and $\mathtt{r} \in \mathsf{def}(2)$.

When reasoning recursively, maydef and mayref can be used to reconstruct complete def-use paths and thus deduce the corresponding data dependences. For instance, considering again the example of the sequence presented above in Fig. 10, the data dependencies coming from def-use paths starting in p_1 and ending in p_2 can be obtained from $\mathsf{maydef}\,(p_1)$ and $\mathsf{mayref}\,(p_2)$ associating pairs with matching variables.

The complete definition of the function $\mathsf{data} : \mathit{Prog} \to 2^{L \times L}$ computing the data dependencies is given in Fig. 14. In case of loops, the data dependencies are either the data dependencies coming from inside the body or the dependencies of the condition and of the body on the body itself. The case of the sequence of a statement and a program can be generalized to the sequence of two programs. The corresponding equation is:

$$\mathsf{data}\,(p_1;p_2) = \mathsf{data}\,(p_1) \cup \mathsf{data}\,(p_2) \cup \{(l_1,l_2) \mid \exists x, (x,l_1) \in \mathsf{maydef}\,(p_1) \wedge (x,l_2) \in \mathsf{mayref}\,(p_2)\}$$

In the example of Fig. 1a, we deduce from $\mathsf{maydef}\,(p_1)$ and $\mathsf{mayref}\,(p_2)$ the two dependencies (2,6) and (5,6). Indeed, the complete def-use paths (required by Def. 5.3) correspond to the concatenations of the paths 1, 2, 3 and 1, 2, 3, 4, 5, 3 of p_1 with the path 6, 7 of p_2.

Theorem 7.1 states the correctness of the assert, control and data functions with respect to Def. 5.5, 5.1 and 5.3, respectively.

Theorem 7.1 (correctness of assert, control and data)**.** *Let p be a program, and (l, l') a pair of labels in p. Then*

- $(l,l') \in \mathsf{assert}\,(p)$ *if and only if* $l \xrightarrow[p]{\mathcal{D}_a} l'$,
- $(l,l') \in \mathsf{control}\,(p)$ *if and only if* $l \xrightarrow[p]{\mathcal{D}_c} l'$,
- $(l,l') \in \mathsf{data}\,(p)$ *if and only if* $l \xrightarrow[p]{\mathcal{D}_d} l'$.

Proof. The proof of the first two statements is straightforward and can be found in the Coq development. We show here the last statement only.

Let us first prove the implication from left to right. Assume that $(l,l') \in \mathsf{data}\,(p)$. We proceed by structural induction over p.

- For the empty program, $\mathsf{data}\,(\lambda) = \varnothing$. That contradicts the assumption, so the desired implication holds.
- For a sequence of statements:

  $$\mathsf{data}\,(s;p) = \mathsf{data}\,(s) \cup \mathsf{data}\,(p) \cup \{(l_1,l_2) \mid \exists x, (x,l_1) \in \mathsf{maydef}\,(s) \wedge (x,l_2) \in \mathsf{mayref}\,(p)\}$$

  It follows that either $(l,l') \in \mathsf{data}\,(s)$, or $(l,l') \in \mathsf{data}\,(p)$, or there exists x such that $(x,l) \in \mathsf{maydef}\,(s)$ and $(x,l') \in \mathsf{mayref}\,(p)$. The first two cases are almost immediate by induction. Indeed, by induction hypothesis, in this case we have $l \xrightarrow[s]{\mathcal{D}_d} l'$ (resp., $l \xrightarrow[p]{\mathcal{D}_d} l'$), so by Def. 5.3 there is a def-use path inside s (resp., inside p) that we can simply complete to build a suitable def-use path of $s;p$. Therefore, $l \xrightarrow[s;p]{\mathcal{D}_d} l'$.

The third case corresponds to the traversing path in Fig. 10. By Lemma 7.1(ii), there exists a path $\pi_1 l \pi_2$ of s such that $x \in \mathsf{def}(l)$ and for all $l'' \in \pi_2$, we have $x \notin \mathsf{def}(l'')$. By Lemma 7.1(iii), there exists a path $\rho_1 l' \rho_2$ of p such that $x \in \mathsf{ref}(l')$ and for all $l'' \in \rho_1$, we have $x \notin \mathsf{def}(l'')$. If we concatenate the two paths, we obtain $\pi_1 l \pi_2 \rho_1 l' \rho 2$, a syntactic path of $s; p$, where $x \in \mathsf{def}(l) \cap \mathsf{ref}(l')$ and for all $l'' \in \pi_2 \rho_1, x \notin \mathsf{def}(l'')$. Thus $l \xrightarrow[s;p]{\mathcal{D}_d} l'$ by Def. 5.3.

- For a **skip** statement, an assignment or an assertion s, $\mathsf{data}\,(s) = \varnothing$, that contradicts the assumption.
- For an **if** statement, $\mathsf{data}\,(\texttt{if}\ (l'' : b)\ p\ \texttt{else}\ q) = \mathsf{data}\,(p) \cup \mathsf{data}\,(q)$. Thus, we have either $(l, l') \in \mathsf{data}\,(p)$ or $(l, l') \in \mathsf{data}\,(q)$. By the induction hypothesis, we deduce either $l \xrightarrow[p]{\mathcal{D}_d} l'$ or $l \xrightarrow[q]{\mathcal{D}_d} l'$. In each case, by prepending l'' to the corresponding syntactic path, we get a path of $\texttt{if}\ (l'' : b)\ p\ \texttt{else}\ q$ showing that $l \xrightarrow[\texttt{if}\ (l'':b)\ p\ \texttt{else}\ q]{\mathcal{D}_d} l'$.
- For a **while** statement:

  $$\mathsf{data}\,(\texttt{while}\ (l'' : b)\ p) = \mathsf{data}\,(p) \cup \{(l_1, l_2) \mid \exists x, (x, l_1) \in \mathsf{maydef}\,(p) \wedge (x, l_2) \in \mathsf{mayref}\,(\texttt{while}\ (l'' : b)\ p)\}$$

  It follows that either $(l, l') \in \mathsf{data}\,(p)$, or there exists x such that $(x, l) \in \mathsf{maydef}\,(p)$ and $(x, l') \in \mathsf{mayref}\,(\texttt{while}\ (l'' : b)\ p)$. In the first case, like in the **if** case, we deduce $l \xrightarrow[p]{\mathcal{D}_d} l'$, and by prepending and appending l'' to the corresponding syntactic path, we get a path of $\texttt{while}\ (l'' : b)\ p$ (corresponding to one iteration of the loop) showing that $l \xrightarrow[\texttt{while}\ (l'':b)\ p]{\mathcal{D}_d} l'$. In the second case, by Lemma 7.1(ii), there exists a path $\pi_1 l \pi_2$ of p such that $x \in \mathsf{def}(l)$ and for all $l''' \in \pi_2$, we have $x \notin \mathsf{def}(l''')$. By Lemma 7.1(iii), there exists a path $\rho_1 l' \rho_2$ of $\texttt{while}\ (l'' : b)\ p$ such that $x \in \mathsf{ref}(l')$ and for all $l''' \in \rho_1$, we have $x \notin def(l''')$. If we concatenate the two paths and prepend l'' to the result, we get $l'' \pi_1 l \pi_2 \rho_1 l' \rho_2$, a syntactic path of $\texttt{while}\ (l'' : b)\ p$, where $x \in \mathsf{def}(l) \cap \mathsf{ref}(l')$ and for all $l''' \in \pi_2 \rho_1, x \notin \mathsf{def}(l''')$. Therefore $l \xrightarrow[\texttt{while}\ (l'':b)\ p]{\mathcal{D}_d} l'$.

Let us now give a sketch of proof for the implication from right to left. This proof is more technical, so we give here the key ideas and describe the main steps. (The detailed proof can be found in the Coq development.) Assume that $l \xrightarrow[p]{\mathcal{D}_d} l'$. By Def. 5.3, there exist a syntactic path $\pi_1 l \pi_2 l' \pi_3$ of p and a variable x such that $x \in \mathsf{def}(l) \cap \mathsf{ref}(l')$ and for all $l'' \in \pi_2$, we have $x \notin \mathsf{def}(l'')$.

To perform the proof, we need to introduce the following path-oriented versions of the functions defined in Fig. 11, 12, 13 and 14, in which the corresponding computation is reduced to a given syntactic path. The empty path is denoted by ".".

$$\begin{array}{llll}
\mathsf{mustdef}_p(.) = \varnothing, & \mathsf{mustdef}_p(l_1\pi) & = \mathsf{def}(l_1) \cup \mathsf{mustdef}_p(\pi) \\
\mathsf{maydef}_p(.) = \varnothing, & \mathsf{maydef}_p(l_1\pi) & = \{(x, l_1) \mid x \in \mathsf{def}(l_1) \wedge x \notin \mathsf{mustdef}_p(\pi)\} \cup \mathsf{maydef}_p(\pi) \\
\mathsf{mayref}_p(.) = \varnothing, & \mathsf{mayref}_p(l_1\pi) & = \{(x, l_1) \mid x \in \mathsf{ref}(l_1)\} \cup \{(x, l_1) \mid x \in \mathsf{mayref}_p(\pi) \wedge x \notin \mathsf{mustdef}_p\,(\pi)\} \\
\mathsf{data}_p(.) = \varnothing, & \mathsf{data}_p(l_1\pi) & = \{(l_1, l_2) \mid \exists x, x \in \mathsf{def}(l_1) \wedge (x, l_2) \in \mathsf{mayref}_p(\pi)\} \cup \mathsf{data}_p\,(\pi)
\end{array}$$

Intuitively, $\mathsf{data}_p(\pi)$ computes the set of data dependencies that can be found along the given path π. It is thus easily seen that for any syntactic path π of p, $\mathsf{data}_p(\pi) \subseteq \mathsf{data}\,(p)$. This fact is the key argument in the remainder of the proof.

We therefore have $\mathsf{data}_p(\pi_1 l \pi_2 l' \pi_3) \subseteq \mathsf{data}\,(p)$, and we have to show that $(l, l') \in \mathsf{data}\,(p)$. It is thus sufficient to prove that $(l, l') \in \mathsf{data}_p(\pi_1 l \pi_2 l' \pi_3)$. Using recursively the relations presented above, we can deduce the following property:

$$\mathsf{data}_p(\pi_1 l \pi_2 l' \pi_3) = \mathsf{data}_p(\pi_1) \cup \mathsf{data}_p(l \pi_2 l' \pi_3) \cup \{(l'', l''') \mid \exists x, (x, l'') \in \mathsf{maydef}_p(\pi_1) \wedge (x, l''') \in \mathsf{mayref}_p(l \pi_2 l' \pi_3)\}$$

where $\mathsf{data}_p(l \pi_2 l' \pi_3) = \{(l, l'') \mid \exists y, y \in \mathsf{def}(l) \wedge (y, l'') \in \mathsf{mayref}_p(\pi_2 l' \pi_3)\} \cup \mathsf{data}_p\,(\pi_2 l' \pi_3)$. We claim that (l, l') belongs to the left subset of this union, and consider $y = x$. Since $x \in \mathsf{def}(l)$ holds by assumption, it remains to show that $(x, l') \in \mathsf{mayref}_p(\pi_2 l' \pi_3)$. Using recursively the definitions, we can deduce:

$$\mathsf{mayref}_p(\pi_2 l' \pi_3) = \mathsf{mayref}_p(\pi_2) \cup \{(x, l'') \mid (x, l'') \in \mathsf{mayref}_p(l' \pi_3) \wedge x \notin \mathsf{mustdef}_p(\pi_2)\}.$$

```
Inductive prog : Type :=
  | P_nil : prog (* the empty program *)
  | P_cons : stmt → prog → prog (* a statement plus a program *)
with stmt :=
  | S_skip : label → stmt (* skip *)
  | S_ass : label → id → aexp → stmt (* assignment *)
  | S_if : label → bexp → prog → prog → stmt (* if *)
  | S_while : label → bexp → prog → stmt (* while *)
  | S_assert : label → bexp → label → stmt. (* assertion *)
```

Fig. 15. Language definition

We claim that the right subset contains (x, l'). To show that $(x, l') \in \mathsf{mayref}_p(l'\pi_3)$, notice that, since $x \in \mathsf{ref}(l')$ by assumption, $(x, l') \in \{(y, l') \mid y \in \mathsf{ref}(l')\} \subseteq \mathsf{mayref}_p(l'\pi_3)$. Furthermore, by assumption, for all $l'' \in \pi_2$, we have $x \notin \mathsf{def}(l'')$, thus, by definition, $x \notin \mathsf{mustdef}_p(\pi_2)$. We conclude that $(l, l') \in \mathsf{data}_p(\pi_1 l \pi_2 l' \pi_3)$, and therefore $(l, l') \in \mathsf{data}\,(p)$, which ends this proof. □

Theorem 7.1 shows that assert, control and data correctly implement the computation of the corresponding dependencies. It will be used in the Coq formalization presented hereafter.

7.2. Overview of the Coq Formalization

Language. The WHILE language defined in Coq is very similar to the one given in Sec. 4, but slightly less expressive. In particular, to fulfill the hypothesis that all potential errors are protected by assertions, possibly erroneous elements such as division and arrays are not included in the language. In this language, assertions are really the only sources of failures. This is the simplest language with errors and nontermination that remains representative for our purpose.

The definition is quite standard and readable even for those who are not familiar with Coq. It is given in Fig. 15. In this definition, `id` and `label` are basically `nat`, the type of natural integers, and `aexp` (resp. `bexp`) is the type of arithmetic (resp. boolean) expressions. The definition of the language and most of the basic notions are strongly inspired by a Coq tutorial [PCG+15].

As it can be seen in the definition, all statements are labelled, and assertions have a second label to point to other statements. But nothing ensures here that the labels of different statements are distinct (contrary to the paper-and-pencil formalization in Sec. 4–6, where this assumption was very convenient), and this will require some care in the remainder of the development, since a label cannot be used to uniquely identify a statement of the program.

Semantics. The normal states of a program are expressed as functions from identifiers to values (of type `state := id → nat`). A state is of type `state_eps := option state`, i.e. it is either `None` (the error state) or `Some st`, where `st` is a normal state.

The denotational semantics is given as a recursive function

```
traj_prog : nat → state_eps → prog → traj
```

taking as parameters a desired number of trajectory steps, an initial state and a program, and returning a finite prefix of the full trajectory of the program from this initial state (of type `traj := list (label * state_eps)`).

A similar function

```
proj_traj_prog : nat → state_eps → prog → set label → partial_traj
```

takes as an additional parameter a set of labels and computes the projected trajectory to this set. Its return type is `partial_traj = list (label * partial_state_eps)`, where `partial_state_eps` is either the error state or a partially defined state. Note that this function cannot be implemented using `traj_prog`, due to the non-unicity of labels.

Dependences. The dependence relations are defined in a similar way as in Sec. 5.

The definition of assertion dependence, shown in Fig. 16, is the simplest one. As Def. 5.5, it connects the

```
Inductive rel_assert : prog → relation label :=
  | ADP_here : forall s p l l', rel_assert_stmt s l l' → rel_assert (P_cons s p) l l'
      (* assertion dependence from inside the first statement *)
  | ADP_after : forall s p l l', rel_assert p l l' → rel_assert (P_cons s p) l l'
      (* assertion dependence from inside the remainder of the program *)
with rel_assert_stmt : stmt → relation label :=
  | ADS_ifb_l : forall l b p1 p2 l' l'', rel_assert p1 l' l'' → rel_assert_stmt (S_if l b p1 p2) l' l''
      (* assertion dependence from inside the then branch *)
  | ADS_ifb_r : forall l b p1 p2 l' l'', rel_assert p2 l' l'' → rel_assert_stmt (S_if l b p1 p2) l' l''
      (* assertion dependence from inside the else branch *)
  | ADS_while : forall l b p l' l'', rel_assert p l' l'' → rel_assert_stmt (S_while l b p) l' l''
      (* assertion dependence from inside the body *)
  | ADS_assert : forall b l l', rel_assert_stmt (S_assert l b l') l l'.
      (* an assertion of label l protecting l' gives the dependency l → l' *)
```

Fig. 16. Implementation of assertion dependence in Coq (cf. Def. 5.5)

```
Inductive rel_control : prog → relation label :=
| CDP_here : ... (* control dependence from inside the first statement *)
| CDP_after : ... (* control dependence from inside the remainder of the program *)
with rel_control_stmt : stmt → relation label :=
| CDS_ifb_l : ... (* control dependence from inside the then branch *)
| CDS_ifb_r : ... (* control dependence from inside the else branch *)
| CDS_while : ... (* control dependence from inside the body *)
| CDS_ifb_cond_l : forall l l' b p1 p2, rel_top_labels p1 l' → rel_control_stmt (S_if l b p1 p2) l l'
    (* control dependencies of (a top-level statement of) the then branch on the condition *)
| CDS_ifb_cond_r : forall l l' b p1 p2, rel_top_labels p2 l' → rel_control_stmt (S_if l b p1 p2) l l'
    (* control dependencies of (a top-level statement of) the else branch on the condition *)
| CDS_while_cond : forall l l' b p, rel_top_labels p l' → rel_control_stmt (S_while l b p) l l'.
    (* control dependencies of (a top-level statement of) the body on the condition *)
```

Fig. 17. Implementation of control dependence in Coq (cf. Def. 5.1)

two labels of an assertion. Most of the cases (`ADP_here`, `ADP_after`, `ADS_ifb_l`, `ADS_ifb_r`, `ADS_while`) explore the program recursively, while the key case, really generating dependencies, is `ADS_assert`.

The definition of (unitary) control dependence is given in Fig. 17 (cf. Def. 5.1). Most of the cases only ensure the recursive descent, as in the definition of `rel_assert`. They are omitted in the figure. Unitary control dependencies can arise from the then branch (`CDS_ifb_cond_l`) or the else branch (`CDS_ifb_cond_r`) of a condition, or the body of a loop (`CDS_while_cond`). Relation `rel_top_labels p l` holds if and only if a statement of label `l` is in the top-level sequence of program `p`, i.e. not in a nested condition or loop. This predicate ensures that `rel_control` characterizes only unitary control dependencies, in the same manner as top_labels in Sec. 7.1. Again, since slicing manipulates reflexive transitive closures of dependence relations, this will not change the slices.

Data dependence, given in Fig. 18, is defined in the same spirit as Def. 5.3. It also uses finite syntactic paths, using two predicates `is_flat_of` and `flat_data`. The predicate `is_flat p q` states that `q` is (a specific representation of) a syntactic path of program `p`, while `flat_data q l l'` states that `q` carries the data dependency of `l'` on `l`. By collecting data dependencies in every syntactic path of `p`, we reconstruct all the data dependencies of `p`.

Slice computation. As mentioned in Sec. 5.1 and commonly done in the litterature, the computation of the slice is divided into two parts: the computation of the set of instructions that must be preserved (commonly called the *slice set*) and the computation of the slice itself from the original program and the slice set.

For the computation of the slice set, we implement the functions control, data and assert discussed in Sec.

```
Inductive rel_data : prog → relation label :=
| DDP_flat : forall p q, is_flat_of q p → forall l l', flat_data q l l' → rel_data p l l'.
    (* a data dependency along a path q of p is a data dependency in p *)
```

Fig. 18. Implementation of data dependences in Coq (cf. Def. 5.3)

```
Fixpoint mustdef p :=
match p with
| P_nil ⇒ ∅
| P_cons s q ⇒ mustdef_stmt s ∪ mustdef q
end
with mustdef_stmt s :=
match s with
| S_skip _ ⇒ ∅
| S_ass _ x _ ⇒ {x}
| S_if _ _ p1 p2 ⇒ mustdef p1 ∩ mustdef p2
| S_while _ _ _ ⇒ ∅
| S_assert _ _ _ ⇒ ∅
end.
```

(a) Definition of mustdef (cf. Fig. 11) in Coq

```
Fixpoint data p :=
  match p with
  | P_nil ⇒ ∅
  | P_cons s q ⇒ data_stmt s ∪ data q ∪
      { (l, l') | ∃ x, (x,l) ∈ maydef_stmt s ∧ (x,l') ∈ mayref q }
  end
with data_stmt s :=
  match s with
  | S_skip _ ⇒ ∅
  | S_ass _ _ _ ⇒ ∅
  | S_if _ _ p1 p2 ⇒ data p1 ∪ data p2
  | S_while _ _ p ⇒ data p ∪
      { (l,l') | ∃ x, (x,l) ∈ maydef p ∧ (x,l') ∈ mayref_stmt s }
  | S_assert _ _ _ ⇒ ∅
  end.
```

(b) Definition of data (cf. Fig. 14) in Coq

Fig. 19. Implementations of mustdef and data in Coq

7.1. data is defined in the same way, using three auxiliary functions: mustdef, maydef and mayref. control, data and assert are used in a function denoted `set_slice : prog → set label → set label`. `set_slice p L` computes the inverse image of the slicing criterion `L` under the reflexive transitive closure of the union of the dependence relations.

As for the second part, the computation of the slice from the slice set, it is implemented by a function called `slice : prog → set label → prog`. `slice p L`, which computes the largest quotient of `p` whose set of labels is `set_slice p L` (due to the non-unicity of labels, a set of labels does not uniquely identify a quotient, we therefore conservatively take the largest). It uses a function `keep_L_prog : prog → set label → set label`, which is the equivalent in Coq of the function $\mathcal{F}_L$ in the proof of Lemma 5.1, and which basically iterates on the statements of the program and removes those whose labels are not to be preserved.

To illustrate the Coq definitions, we present in Fig. 19 the implementations of mustdef and data as Coq functions `mustdef : prog → set id` and `data : prog → set (label * label)`.

Results. The main results, presented in Fig. 20, formalize in Coq the soundness of relaxed slicing (Th. 5.1). The first part of Th. 5.1 is implemented by theorem `slice_proj_prefix` in Fig. 20a. In this theorem, `prefix` formalizes a standard notion of prefix for lists. Thus, the theorem in Fig. 20a expresses in Coq that the projection of the trajectory of the initial program is a prefix of the projection of the trajectory of its slice.

The second theorem (given in Fig. 20b) is more complex due to the fact that we only manipulate trajectories using finite prefixes. The condition `length (traj_prog N0 ste p) < N0` ensures that the execution of `p` stopped before `N0` steps. `last_state (traj_prog N0 ste p) ste <> None` expresses that the last state is not an error. Together, those conditions characterize a normal termination for the execution of `p` from `ste` in less than `N0` steps.

From these theorems, we can prove the formalizations in Coq of the two results of Sec. 6. They can be found in the Coq development.

8. Related Work

Weiser [Wei84] introduced the basics of intraprocedural and interprocedural static slicing. A thorough survey provided in [Tip95] 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 [BH04, Sil12, XQZ+05]. Foundations of program slicing have been studied e.g. in [Amt08, BH93, BBD+10, BDG+06, CF89, DBH+11, HRB88, RAB+07, RY89, RY88]. 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. [ADS93, HHD99, Wei82]) where it can be used to better understand an already detected

```
Theorem slice_proj_prefix :
  forall p L n ste,
  prefix (proj_traj_prog n ste p (set_slice p L))
         (proj_traj_prog n ste (slice p L) (set_slice p L)).
```

(a) Soundness theorem in the general case

```
Theorem slice_proj_equal :
  forall n ste p L,
  length (traj_prog n ste p) < n → (* the execution of p terminates in less than n steps *)
  last_state (traj_prog n ste p) ste <> None → (* the execution of p does not reach an error *)
  length (traj_prog n ste (slice p L)) < n ∧ (* the slice terminates in less then n steps *)
  last_state (traj_prog n ste (slice p L)) ste <> None ∧ (* the slice does not reach an error *)
  proj_traj_prog n ste p (set_slice p L) =
  proj_traj_prog n ste (slice p L) (set_slice p L). (* the projections of the trajectories are equal *)
```

(b) Soundness theorem in the case of normal termination

Fig. 20. Soundness theorem (cf. Th. 5.1) formalized in Coq

error, to prioritize test cases (e.g. in regression testing), simplify a program before testing, etc. In particular, dynamic slicing [BDG+06] 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 [CKGJ12, KKPP15]) in the 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 the presence of non-termination. Among them, we can cite the lazy semantics of [CF89], and the transfinite one of [GM03], improved by [Nes09]. Another semantics proposed in [BBD+10] 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 [CKGJ12, GTXT11].

[RAB+07] 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 [PC90] and further generalized for any finite directed graph in [DBH+11]. [RAB+07] 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.

[Amt08] establishes a soundness property for non-termination insensitive control dependence in terms of simulation. [BH93] describes program slicing for arbitrary control flow. [BH93] and [Amt08] 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 the presence of errors *and* non-termination, explicitly formalizes errors by assertions and deduces several results on performing V&V on slices.

Slicing in the presence of errors. [HSD96] 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 $\mathsf{def}(l)$ and $\mathsf{ref}(l)$ 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. [HD95] 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. [AH03] extends data and control dependences for Java program with exceptions. In both papers, no formal justification is given. Another seemingly related work is [BdCHP12], which focuses on assertion-based slicing. But in this work, the meaning of "assertion" is quite different, since it refers to contracts of functions. Assertion-based slicing is, in their work, another term for specification-based slicing.

Certified slicing. The ideas developed in [Amt08, RAB+07] were applied in [BMP15, Was11]. [Was11]

builds a framework in Isabelle/HOL to formally prove a slicing defined in terms of graphs, therefore language-independent. [BMP15] proposes an unproven but efficient slice calculator for an intermediate language of the CompCert C compiler [Ler09], as well as a certified slice validator and a slice builder written in Coq [BC04]. 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 the 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.

9. 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 the 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 the presence or absence of errors in slices.

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 [Léc16]. This Coq formalization represents an effort of 8 person-months of intensive Coq development resulting in more than 10,000 lines of Coq code.

The current version of the formalization and the slicer strongly depends on the language and its structure. Future work includes an extension to a realistic programming language, a generalization to a wider class of errors 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.

Acknowledgments. 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

[ADS93] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Softw., Pract. Exper.*, 23(6):589–616, 1993.

[AH03] Matthew Allen and Susan Horwitz. Slicing Java programs that throw and catch exceptions. In *PEPM 2003*, pages 44–54, 2003.

[Amt08] Torben Amtoft. Slicing for modern program structures: a theory for eliminating irrelevant loops. *Inf. Process. Lett.*, 106(2):45–51, 2008.

[BBD+10] Richard W. Barraclough, David Binkley, Sebastian Danicic, Mark Harman, Robert M. Hierons, Akos Kiss, Mike Laurence, and Lahcen Ouarbya. A trajectory-based strict semantics for program slicing. *Theor. Comp. Sci.*, 411(11–13):1372–1386, 2010.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development.* Springer, 2004.

[BdCHP12] José Bernardo Barros, Daniela Carneiro da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Assertion-based slicing and slice graphs. *Formal Asp. Comput.*, 24(2):217–248, 2012.

[BDG+06] David Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. Theoretical foundations of dynamic program slicing. *Theor. Comput. Sci.*, 360(1-3):23–41, 2006.

[BH93] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *AADEBUG 1993*, 1993.

[BH04] David Binkley and Mark Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[BMP15] Sandrine Blazy, André Maroneze, and David Pichardie. Verified validation of program slicing. In *CPP 2015*, pages 109–117, 2015.

[CCK+14] O. Chebaro, P. Cuoq, N. Kosmatov, B. Marre, A. Pacalet, N. Williams, and B. Yakobowski. Behind the scenes in SANTE: a combination of static and dynamic analyses. *Autom. Softw. Eng.*, 21(1):107–143, 2014.

[CF89] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. In *PLDI 1989*, 1989.
[CKGJ11] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. The SANTE tool: Value analysis, program slicing and test generation for C program debugging. In *TAP 2011*, 2011.
[CKGJ12] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *SAC 2012*, 2012.
[DBH+11] Sebastian Danicic, Richard W. Barraclough, Mark Harman, John Howroyd, Ákos Kiss, and Michael R. Laurence. A unifying theory of control dependence and its application to arbitrary program structures. *Theor. Comput. Sci.*, 412(49):6809–6842, 2011.
[GM03] Roberto Giacobazzi and Isabella Mastroeni. Non-standard semantics for program slicing. *Higher-Order and Symbolic Computation*, 16(4):297–339, 2003.
[GTXT11] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. DyTa: dynamic symbolic execution guided with static verification results. In *the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 992–994. ACM, 2011.
[HD95] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Softw. Test., Verif. Reliab.*, 5(3):143–162, 1995.
[HHD99] Robert M. Hierons, Mark Harman, and Sebastian Danicic. Using program slicing to assist in the detection of equivalent mutants. *Softw. Test., Verif. Reliab.*, 9(4):233–262, 1999.
[HRB88] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *PLDI 1988*, 1988.
[HSD96] Mark Harman, Dan Simpson, and Sebastian Danicic. Slicing programs in the presence of errors. *Formal Aspects of Computing*, 8(4):490–497, 1996.
[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.
[KKPP15] Balázs Kiss, Nikolai Kosmatov, Dillon Pariente, and Armand Puccetti. Combining static and dynamic analyses for vulnerability detection: Illustration on Heartbleed. In *HVC 2015*, 2015.
[Léc16] Jean-Christophe Léchenet. Formalization of relaxed slicing, 2016. `http://perso.ecp.fr/~lechenetjc/slicing/`.
[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
[LKG16] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. Cut branches before looking for bugs: Sound verification on relaxed slices. In *FASE'16 (Part of ETAPS'16)*, pages 179–196, 2016.
[Nes09] Härmel Nestra. Transfinite semantics in the form of greatest fixpoint. *J. Log. Algebr. Program.*, 78(7):573–592, 2009.
[PC90] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Software Eng.*, 16(9):965–979, 1990.
[PCG+15] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations 3.2, 2015. `http://www.cis.upenn.edu/~bcpierce/sf/sf-3.2/index.html`.
[RAB+07] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.
[RY88] Thomas W. Reps and Wuu Yang. The semantics of program slicing. Technical report, Univ. of Wisconsin, 1988.
[RY89] Thomas W. Reps and Wuu Yang. The semantics of program slicing and program integration. In *TAPSOFT 1989*, 1989.
[Sil12] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):12, 2012.
[Tip95] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.
[Was11] Daniel Wasserrab. *From formal semantics to verified slicing: a modular framework with applications in language based security.* PhD thesis, Karlsruhe Inst. of Techn., 2011.
[Wei81] Mark Weiser. Program slicing. In *ICSE 1981*, 1981.
[Wei82] Mark Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.
[Wei84] Mark Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.
[XQZ+05] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.