Program slicing techniques with support for unconditional jumps [⋆]

Carlos Galindo[0000−0002−3569−6218], Sergio Pérez[0000−0002−4384−7004], and Josep Silva[0000−0001−5096−0008]

VRAIN, Universitat Politècnica de València,
Camino de Vera s/n, 46022, Valencia, Spain
`cargaji@vrain.upv.es` `{serperu,jsilva}@dsic.upv.es`

Abstract. The System Dependence Graph is a data structure often used in software analysis, and in particular in program slicing. Multiple optimizations and extensions have been created to give support to common language features, such as unconditional jumps (PPDG) and object-oriented features (JSysDG). In this paper we show that, unfortunately, the solutions proposed for different problems are incompatible when they are combined, producing incorrect results. We identify and showcase the incompatibilities generated by the combination of different techniques, and propose specific solutions for every incompatibility described. Finally, we present an implementation in which the issues found have been addressed, producing correct slices in all cases.

Keywords: Program slicing · Pseudo-predicate program dependence graph · Control dependence · System dependence graph

1 Introduction

Program slicing [20,23] is a software analysis technique that extracts the subset of program instructions, the *program slice* [25], that affect or are affected by a specific program point called *slicing criterion*. Program slicing has a wide range of applications such as software maintenance [22], debugging [6], code obfuscation [18], program specialization [19], and even artificial intelligence [26]. Most program slicing techniques are based on a program graph representation called the System Dependence Graph (SDG). The SDG [11] is a directed graph that represents statements as nodes and their dependences as arcs. It builds upon and is composed of Program Dependence Graphs (PDG) [7], a similar intraprocedural graph.

[⋆] This work has been partially supported by grant PID2019-104735RB-C41 funded by MCIN/AEI/ 10.13039/501100011033, by the *Generalitat Valenciana* under grant Prometeo/2019/098 (DeepTrust), and by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215. Sergio Pérez was partially supported by *Universitat Politècnica de València* under FPI grant PAID-01-18. Carlos Galindo was partially supported by the Spanish *Ministerio de Universidades* under grant FPU20/03861.

Example 1 (Program slicing). Consider the program in Figure 1a. If we are interested in determining what parts of the program are needed to compute the value of variable `min` at line 12 we can define the slicing criterion $\langle 12, min \rangle$ (blue code in Figure 1a). A program slicer will automatically generate the program representation (the SDG) and use the standard slicing algorithm (presented in [11]) to compute the slice shown in Figure 1b, where all the statements that cannot influence the value of variable min have been removed from the program.

```
1       min_max(int[] n) {
2       min = n[0];
3       max = n[0];
4       i = 1;
5       while (i < n.length) {
6         if (a[i] < min)
7           min = a[i];
8         if (a[i] > max)
9           max = a[i];
10        i++;
11      }
12      print(min);
13      print(max);
14  }
```

(a) Program to compute min and max.

```
1       min_max(int[] n) {
2       min = n[0];
3
4       i = 1;
5       while (i < n.length) {
6         if (a[i] < min)
7           min = a[i];
8
9
10        i++;
11      }
12      print(min);
13
14  }
```

(b) Slice w.r.t. $\langle 12, min \rangle$.

Fig. 1: Example of program slicing.

The original formulation of the SDG covered a simple imperative language with branches, loops, and calls; but later research have proposed extensions to cover different constructs of programming languages from all programming paradigms. Currently, there are versions of the SDG that cover most syntax constructs of some languages such as Java[1], C and C++[2], Erlang[3], or Javascript [21], among many others. Additionally, there are techniques that increase the precision of the slices generated, by removing or limiting the traversal of some dependences (see, e.g., [15,9,8,14]).

There are three important extensions of the SDG that we explore in this paper: (1) the extension of the SDG for object-oriented programs (the JSysDG [24]), the exception handling extension [9] (treatment for *try-catch*, *throws*...), and the unconditional jumps extension [15] (treatment for *break*, *continue*, *return*...). The three extensions have been implemented and work for different languages. Nevertheless, we are not aware of any implementation that combines all of them.

With the aim of developing a program slicer that is able to treat the whole Java language, we have implemented several extensions including those three.

[1] Codesonar SAST: `https://www.grammatech.com/codesonar-sast-java`
[2] Codesonar: `https://www.grammatech.com/codesonar-cc`
[3] e-Knife: `https://mist.dsic.upv.es/e-knife`

During the implementation and testing processes, we discovered a set of collisions between the extension of the PDG given to treat unconditional jumps (the pseudo-predicate program dependence graph (PPDG) [15]) and the representation models used to solve other language features, leading, in several cases, to erroneous slices. In this paper, we showcase a set of identified incompatibilities generated by the combination of the PPDG representation used to treat unconditional jumps with other representation models used to solve orthogonal problems such as call representation, object-oriented features, or exception-handling. In most cases these incompatibilities produce an incorrect use of control-dependence arcs. For each detected conflict, we explain the rationale behind it, and propose a specific change in the model to solve it. This is valuable information for developers of program slicers and essential information for the definition of a more general theoretical program slicing model. From the practical perspective, we also provide an open-source implementation of a program slicer for a wide subset of Java.

The main contributions of this paper can be summarized as follows:

- The identification of several state-of-the-art program slicing techniques that cannot be implemented together. A counterexample is presented for each incompatible combination.
- The theoretical explanation about why each technique is incompatible with the PPDG and a solution proposed for each problem identified.
- The implementation of a public open-source program slicer for Java that implements all the techniques discussed in the paper with the solutions proposed, which makes them compatible.

The paper is structured as follows: Section 2 briefly explains how an SDG is built; Section 3 introduces the PPDG, a modification to the PDG to support unconditional jumps; Section 4 showcases the interference between the PPDG and techniques that handle calls, exceptions, and object-oriented programs; Section 5 describes the implementation where this interference has been corrected; Section 6 shows the related work; and Section 7 presents our conclusions.

2 Background

To keep this paper self-contained, in this section we introduce the necessary background on program slicing and the SDG. Readers familiar with program slicing can skip this section.

The SDG is built from a sequence of graphs: a control-flow graph (CFG) [1] is computed from the source code of each procedure in the program. Then, control and data dependences are computed in each CFG to create a series of Program Dependence Graphs (PDG); and, finally, the calls in the PDGs are connected to their corresponding procedures to create the System Dependence Graph (SDG). A slice can then be computed by traversing the SDG backwards from the node that represents the slicing criterion, producing the subset of nodes that affect it [11].

The CFG is a representation of the execution flow of a procedure in a program. Each statement is represented as a node (in a set N), and each arc (in a set A) represents the sequential execution of a pair of statements. In the CFG, statements only have one outgoing control-flow arc, and predicates, have a set of outgoing arcs. Additionally, the CFG contains two extra nodes, "Enter" and "Exit", which represent the source and sink of the graph. If a procedure has multiple `return` statements, then all of them are connected to the exit node.

The PDG [7] is built by computing control and data dependences (Definitions 2 and 3): given a CFG $G = (N, A)$, its corresponding PDG is $G' = (N \setminus \{Exit\}, A_c \cup A_d)$, where A_c and A_d are the set of control and data dependences, represented as directed arcs ($a \to b$ iff b is dependent on a).

Definition 1 (Postdominance [23]). *Given a CFG $G = (N, A)$, a node $m \in N$ postdominates a node $n \in N$ if and only if all paths from n to the "Exit" node in G contain m.*

Definition 2 (Control dependence [10]). *Given a CFG $G = (N, A)$, a node $b \in N$ is* control dependent *on a node $a \in N$ if and only if b postdominates some but not all of a's successors in G.*

Definition 3 (Data dependence [10]). *Given a CFG $G = (N, A)$, a node $n \in N$ is* data dependent *on a node $m \in N$ if and only if m defines (i.e., assigns a value to) a variable x, n uses variable x and there is a path in G from m to n where x is not redefined.*

The SDG [11] is the union of multiple PDGs, with the addition of new dependence arcs that connect procedure calls to their respective procedure definitions. Each node that contains a call is split into several nodes. For every call, there is a node that represents the call itself, and one node per input and output. These nodes are inserted in the graph, and connected to the node that contains the call via control dependence arcs. A similar structure is generated by the "Enter" node of each procedure definition: the inputs and outputs of a procedure are placed as nodes that are control dependent on the "Enter" node. When building the SDG, each call is connected to the "Enter" node of their target procedure, each input from a call is connected to a procedure's input, and each procedure's output is connected to the call's output via interprocedural arcs (input and output arcs, respectively).

Finally, summary arcs are added to method calls. Summary arcs join call inputs and call outputs when the value of the input is required inside the method definition to compute the output's value. The following example builds an SDG illustrating the whole process:

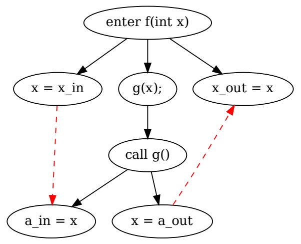
Example 2 (The creation of an SDG for a simple program). Consider the simple program shown in Figure 2a. This program contains two procedures, `f` and `g`, where `f` calls `g` following the call-by-reference method. The computation of the CFG and the application of Definitions 2 and 3 for procedure `f` result in the PDG shown in Figure 2b, where black arcs represent control dependences and red dashed arcs data dependences. Note that, for call nodes, inputs and outputs are separated from the call node and inserted in the PDG. Finally, the SDG is created by combining the PDG for each procedure and inserting interprocedural (blue and dotted) and summary (green, dashed and bold) arcs. The result is shown in Figure 2c.

Given a SDG, a slice can be generated by traversing the arcs backwards, starting on the node that contains the slicing criterion. There are two phases: in the first, the arcs are traversed from the slicing criterion, ignoring output arcs (between output variables); and once no more arcs can be traversed, the second phase begins, starting with all the nodes reached in the first, but ignoring input arcs (between calls and definitions or input variables). The set of nodes reached at the end of the second phase is the slice.

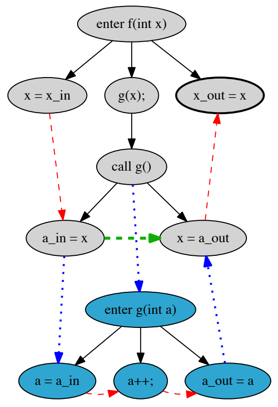
Example 3 (Slicing the SDG in two phases). Consider the SDG in Figure 2c. If node $x_{out} = x$ is picked as slicing criterion (it is shown in bold in the graph, and it represents the value of x at the end of `f(x)`), all the nodes from procedure `f` are included in the first phase (shown in grey). However, the nodes from `g` cannot be reached in this phase, as the $a_{out} = a \rightarrow x = a_{out}$ output arc (lower-right corner) cannot be traversed during the first phase. In the second phase it is traversed, and all the nodes that form `g` are added to the slice (shown in blue). The result is that all nodes are relevant to the value of x at the end of `f(x)`.

```
void f(int x) {
  g(x);
}
void g(int a) {
  a++;
}
```

(a) A program with two simple procedures.



(b) PDG for `f`.



(c) SDG for the whole program and slice w.r.t. $\texttt{x}_{out}$ `= x` node.

Fig. 2: The creation of the SDG for a small program with two procedures.

3 The Pseudo-predicate Program Dependence Graph

The *Pseudo-predicate Program Dependence Graph* (PPDG) [15] is an extension of the original PDG, which includes support for all kinds of unconditional jumps, such as *break*, *goto*, *return*, or *throw*. It minimizes the number of unconditional jumps that are included in slices, with special effectiveness in structures like *switch*, as each case is typically terminated with a *break* or *return*.

The PPDG is built upon the Augmented Control-Flow Graph (ACFG) [2], which introduced a mechanism to correctly generate the control dependences caused by unconditional jumps. In the ACFG, unconditional jumps are not considered statements (which unconditionally execute the next instruction), neither predicates (which branch into multiple possible execution paths), they are classified as a new category of instruction called pseudo-predicate. As it happens with predicates, pseudo-predicates are represented as a node with two outgoing arcs: a true arc pointing to the destination of the unconditional jump and a false arc (called non-executable branch) pointing to the statement that would be executed if the statement failed to jump.

The PPDG's main contribution is two-fold: it modifies the definition of control dependence (replacing Definition 2 with Definition 4), and adds a traversal restriction to the slicing algorithm (see Definition 5) for control dependence arcs.

Definition 4 (Control dependence in the presence of pseudo-predicates [15]). *Let $G = (N, A)$ be a CFG, and let $G' = (N, A')$ with $A \subseteq A'$ be an ACFG, both representing the same procedure. A node $b \in N$ is control-dependent on node $a \in N$ if and only if b postdominates in G some but not all successors of a in G'.*

Definition 5 (PPDG traversal limitations [15]). *Given a PPDG $G = (N, A)$, a pseudo-predicate node $b \in N$ and a control dependence arc $(a, b) \in A$, if b is included in the slice, a should not be included in the slice if:*

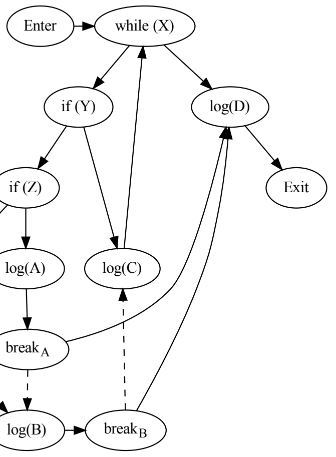
1. *b is not the slicing criterion, and*
2. *b has only been reached via control dependence arcs.*

These two changes are interconnected: Definition 4 generates additional control dependence arcs, which explicitly represent part of the transitive control dependence in the graph; while the restriction of Definition 5 removes part of that transitivity during the slicing traversal of the graph. Together, they act upon pseudo-predicate nodes, maintaining or reducing the amount of pseudo-predicates included in slices (w.r.t. to the PDG built from the ACFG).

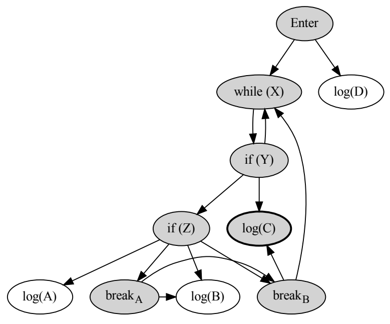
Example 4. Consider the code in Figure 3a, in which a simple loop may terminate via any of the two `break` instructions. Figure 3b shows its CFG, where both `break`s are represented as pseudo-predicates (the dashed edges are non-executable control-flow arcs). According to Definition 2 the PDG of this code is shown in Figure 3c. Similarly, using Definition 4, the PPDG of this code is the one in (Figure 3d). In this specific case, the PDG and PPDG match. The difference between them can be seen in the slices they produce. If we pick $\langle 11, \texttt{C} \rangle$ as the slicing criterion, the PPDG obtains a better result, as it excludes `if (Z)` and $\texttt{break}_A$. The cause of this discrepancy is the traversal limitation established by Definition 5, by which once the slice has reached $\texttt{break}_B$, it cannot continue, as it is a pseudo-predicate, it is not the slicing criterion, and it has only been reached by control dependences.

```
 1  void main() {
 2      while (X) {
 3          if (Y) {
 4              if (Z) {
 5                  log(A);
 6                  break;
 7              }
 8              log(B);
 9              break;
10          }
11          log(C);
12      }
13      log(D);
14  }
```

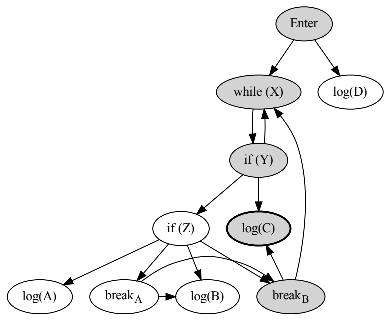
(a) A simple program with unconditional jumps.



(b) The CFG.



(c) The PDG with the slice shown in grey.



(d) The PPDG with the slice shown in grey.

Fig. 3: A comparison between the PDG and the PPDG.

Due to the increase in precision regarding unconditional jumps, the PPDG should be considered the baseline for adaptations of the PDG. However, the balance set by Definitions 4 and 5 is delicate, as it assumes that:

Assumption $\mathcal{A}$: all control dependence arcs in the SDG are generated according to Definition 4.

4 Program slicing techniques that conflict with the PPDG

In the following sections, we showcase examples where the combination of the PPDG with another slicing technique results in Assumption $\mathcal{A}$ breaking down, and thus produces incorrect slices.

4.1 Representation of procedure calls

The sequence of graphs used in program slicing (CFG, PDG, SDG) represent most statements as a single node. Some extensions generate additional nodes to represent other elements, but the most common is the representation of calls and their arguments [11].

As described at the end of Section 2, calls are represented as an additional node in the PDG and SDG, connected to the node they belong to via a control dependence arc. It is precisely the insertion of that arc that breaks $\mathcal{A}$, because it has been hand-inserted instead of generated from Definition 4. Example 5 provides a specific counter-example, where these additional control dependence arcs exclude the "Enter" node from the slice.

Example 5 (Erroneous program slicing of procedure calls in the PPDG). Consider the code and associated SDG shown in Figures 4a and 4b. It contains two procedures, `main` and `f`, with one statement each. If the SDG is sliced w.r.t. the `log` statement, the traversal limitation outlined in Definition 5 will stop the traversal in the arc connecting "Enter `main()`" and "`return f()`" because the second one is a pseudo-predicate. Thus, code that should belong to the slice is excluded and an erroneous slice is produced.

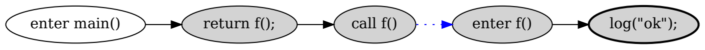
The source of the problem is the incorrect use of control arcs to connect nodes with a call (`return f()`) with the call node (`call f()`). This happens in most nodes that are split in order to improve precision: a control arc is used as a default kind of arc.

The solution would be to change the kind of arc used to connect these kinds of nodes (call nodes, argument and parameter input/output...) from control arc into any other kind. A data arc would not be semantically appropriate, so our proposal is the introduction of what we call *structural arcs*. Formally,

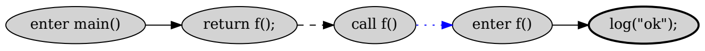
Definition 6 (Structural Arc). *Given two PDG nodes $n, n' \in N$, there exists a* structural edge $n \dashrightarrow n'$ *if and only if:*

```
int main() {
  return f();
}
int f() {
  log("ok");
}
```

(a) A program with two procedures.



(b) The SDG for the program.



(c) The modified SDG that produces the correct slice.

Fig. 4: A simple SDG with two procedures (`main` and `f`), sliced w.r.t. the `log` statement. The slice is represented by grey nodes, the dotted blue arc is an interprocedural call arc, the black solid arcs represent control dependence arcs, and the dashed black arc is a structural arc.

- *n' contains a subexpression from n, and*
- *$\forall n'' \in N$: if $n \dashrightarrow n' \wedge n' \dashrightarrow n'' \Rightarrow n \not\dashrightarrow n''$ (structural edges define an intransitive relation).*

Structural arcs have no restriction on their traversal, and otherwise behave as control arcs before the introduction of the PPDG's traversal restrictions.

An example of this solution has been applied to the erroneous Figure 4b, producing Figure 4c, a SDG where the traversal completes as expected.

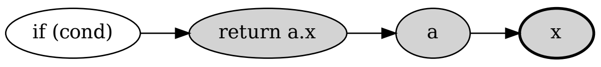
4.2 Object-oriented program slicing

Object-oriented programming languages are widely used, and there are program slicing techniques that adapt the SDG to include features such as polymorphism and inheritance [17]. An example is the *Java System Dependence Graph* (JSysDG), proposed by Walkinshaw et al. [24], an extension built upon the ClDG [16] (a previous attempt to include classes/objects in the SDG). Among other modifications, the JSysDG represents variables with fields as a tree of nodes. This allows the representation of polymorphism and increases its precision, allowing the selection of single fields.

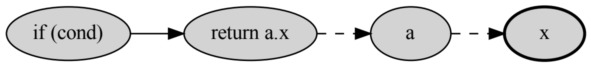
Figure 5a showcases the structure of an object `a`, which contains a field `x`. As with procedure calls described in Section 4.1, the usage of control dependence arcs to represent the structure of the objects violates Assumption $\mathcal{A}$. The same solution can be applied here: using a different kind of arc to represent object structures.

Example 6 (Fixing the "JSysDG with unconditional jumps"). Consider again the graph in Figure 5a, which has been transformed to produce Figure 5b: some arcs have been turned into structural arcs (applying Definition 6). Now, the remaining control arc (between `if` and `return`) can be traversed by the standard algorithm and all the code is correctly included in the slice.

```
if (cond)
  return a.x;
```



(a) The original JSysDG.



(b) The modified JSysDG that produces the correct slice.

Fig. 5: A segment of a JSysDG and its corresponding code snippet. A slice has been produced, w.r.t. variable `a.x` in the `return` statement.

4.3 Exception handling and conditional control dependence

The introduction of exception handling, including instructions like `throw`, `try`, `catch`, and `finally`, motivates the addition of a new kind of control dependence to increase the precision of `catch` statements: *conditional control dependence* [9].

Definition 7 (Conditional control dependence [9]). *Let $P = (N, A)$ be a PDG. We say that a node $b \in N$ is conditional control dependent on a pair of nodes $a, c \in N$, if:*

- *a is a pseudo-predicate,*
- *a controls b, and*
- *c is dependent on b.*

This definition is required to properly model the need for `catch` statements, as can be seen in the following example.

Example 7 (Conditional control dependence with `catch` statements). Consider three instructions a, b, and c where:

- a produces exceptions (e.g. `throw e`),
- b catches those exceptions (e.g. `catch (...) {}`) and
- c only executes if the exception is caught.

In this case, b is needed if and only if both a and c are present in the slice. Otherwise, either the exception is not thrown (a is not in the slice) or no instruction is affected (c is not in the slice).

Conditional control dependence (CCD) connects three nodes in a relationship where one of them controls the other two, but only if they are both present in the slice. Conditional control dependence is represented in the PPDG and SDG as a pair of arcs, named CC1 and CC2. CC1 and CC2 arcs are generated after the PDG has been built: some control dependence arcs are converted to CC1 arcs and CC2 arcs are inserted.

Similarly to the PPDG, the introduction of CCD requires some changes to the traversal algorithm, which are defined as follows:

Definition 8 (Traversal limitations of CCD). *A SDG that contains conditional control dependence must add the following limitations to its traversal algorithm:*

1. *A node that has only been reached via a CCD arc will not be included in the slice, unless it has been reached by both kinds (CC1 and CC2) of dependency.*
2. *If a node is added to the slice via the previous rule, no arcs are traversed from that node, unless it is reached via a non-CCD arc.*

The introduction of this new dependence and its corresponding arcs break Assumption $\mathcal{A}$ in two different ways: some control dependence arcs have been specialized (CC1), so they are no longer present in the graph; and some control dependence arcs have been inserted (CC2), so they do not follow Definition 4. The consequence is that the slicing traversal of the SDG is stopped too early, and some necessary nodes are left out. This can be in the form of "Enter" nodes, or even the structure that contains it (such as `if` or `for`). Figure 6a shows the described scenario, where the graph traversal stops before reaching the "Enter" node due to the pseudo-predicate nature of the `throw` and `try` nodes.

However, the solution to this situation is not so obvious, as changing the kind of arc does not solve the problem. Instead, one of the traversal restrictions imposed on control dependence arcs must be relaxed.

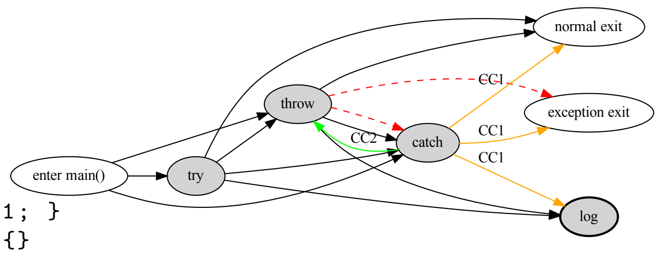
First, the difference between the PPDG and the PDG must be computed. Arcs that are only present in the PPDG and not in the PDG must be marked. Arcs that are marked are not considered control dependences for the purposes of traversal restrictions from the PPDG. This change solves the problem, and the traversal continues normally, reaching the "Enter" node.

Example 8 (Fixing slicing with exceptions and unconditional jumps). Consider Figure 6b, obtained from Figure 6a by marking the control arcs that are exclusive to the PPDG in bold blue. Then, when a slice is computed starting at "log", "try" would be reached via a marked arc, and the traversal would continue back to the "Enter" node, producing a correct slice.

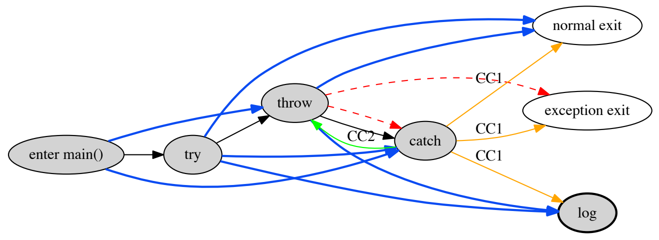
5 Implementation

The incompatibility between different slicing models was detected during the implementation of a slicer that included all the described models. To solve the detected problems, the solutions proposed throughout this paper have been implemented as the base of a new Java slicer: JavaSlicer. It is publicly available

```
void main() {
  try { throw e1; }
  catch (T e2) {}
  log(42);
}
```



(a) The SDG with CCD produces an incorrect slice.



(b) The SDG with CCD and marked arcs produces the correct slice.

Fig. 6: A simple program that throws an exception and two SDGs with CCD, sliced w.r.t. `log`. Black solid arcs are control dependence, red dashed arcs are data dependence, and green and orange solid arcs are conditional control dependence (labelled with their type). Marked control arcs are shown in bold blue.

at `https://mist.dsic.upv.es/git/program-slicing/sdg`, and contains and implements an SDG that covers parts of the Java programming language. It is a project with around 10K lines of code, and it contains a core module that contains the slicer, and a client module, with which users can interact (from the terminal, for now).

Due to space constraints, we have not included in this paper the description of the tests used to validate the proposal presented in this work. However, the interested reader has access to it in the URL: `https://mist.dsic.upv.es/JavaSDGSlicer/evaluation`. There, we describe the suite of tests, and the experiments performed with real programs, showing that all slices produced are equivalent to their associated original programs.

Solving the problems presented in Section 4 only requires changing the kind of arc used (see class `StructuralArc`). However, for Section 4.3 two changes are required. The first is computing the arcs that are present in the PPDG but not in the PDG, which is done in the `PPDG` class:

`PPDG.java`

```java
/** Finds the CD arcs that are only present in the PPDG and marks them as such. */
protected void markPPDGExclusiveEdges(CallableDeclaration<?> declaration) {
    APDG apdg = new APDG();
    apdg.build(declaration);
    Set<Arc> apdgArcs = new HashSet<>();
    for (Arc arc : apgg.edgeSet())
        if (arc.isUnconditionalControlDependencyArc())
            apdgArcs.add(arc);
    for (Arc arc : edgeSet())
        if (arc.isUnconditionalControlDependencyArc()
                && !apdgArcs.contains(arc))
                arc.asControlDependencyArc().setPPDGExclusive();
}
```

Then, in the slicing algorithm, a check is included, to bypass the PPDG traversal condition when an arc is marked. This behaviour is defined in the Java class `ExceptionSensitiveSlicingAlgoritm`, specifically in lines 128-130:

`ExceptionSensitiveSlicingAlgorithm.java`

```java
126 protected boolean essdgIgnore(Arc arc) {
127     GraphNode<?> target = graph.getEdgeTarget(arc);
128     if (arc.isUnconditionalControlDependencyArc() &&
129             arc.asControlDependencyArc().isPPDGExclusive())
130         return false;
131     return hasOnlyBeenReachedBy(target, ConditionalControlDependencyArc.class);
132 }
```

For any interested readers, the software repository contains instructions on how to run the slicer.

6 Related work

There exist very few works that reason about the consequences of integrating different program slicing techniques in the same model. Unfortunately, the solutions proposed for the different slicing problems were mostly proposed in isolation of other problems and solutions. This means that every time a new challenging slicing problem is solved, the solution is presented as an evolution of a previous program representation used to solve the same problem (if there is a previous one) or from the original PDG/SDG representations. For example, this is the case of OO languages where the graph proposed by Larsen and Harrold in [16] was further extended by Liang and Harrold in [17], and lately used by Walkinsaw et al. in [24] to represent Java object-oriented programs. But none of them considered, e.g., exception handling; as they assumed that the standard solution to exception handling would not interfere with their solution to OO. The real world is however a continuous interference between problems and solutions. Any implementation for a real programming language must include different solutions and they will necessarily interfere.

Most solutions proposed in the literature are based on the original SDG and the original slicing algorithm proposed in [11]. Many proposals enhance the expressivity of the graph in different ways: adding modifications to the CFG used to generate the final graph ([15,9]) or including new program dependences

([8,4,5,27], and sometimes these new models come with some modifications to the slicing traversal algorithm to improve computed slices ([15,9,8,14]).

Since the initial graph of all these proposals is the original SDG, most of the solutions designed to slice other previously solved problems are not considered when designing a new program representation model. This is the case of several object-oriented program representations like [16,17], where the SDG is augmented to represent polymorphic method calls with the addition of new graph nodes that consider the statically undecidable method called. Another example is the one presented by Cheng [4,5,27] for program slicing of concurrent programs. Cheng's proposal starts from the PDG and defines a set of specialised dependences required in concurrent environments, building a representation called the *program dependence net* (PDN).

Taking a look at the literature, we consider whether a solution to a particular slicing problem can be classified as control dependence conservative (i.e. when all control arcs in the graph are computed over Definitions 2 and 4) or not. For example, the works presented by Krinke and Snelting [13], Cheda et al. [3], and the one presented by Kinloch and Munro [12] are all control dependence conservative. While Krinke and Snelting, and Cheda et al. define a new type of arc to represent what we call structural dependences, Kinloch and Munro construct the PDG by splitting assignments in different nodes (left-hand side and right-hand side) controlled by the same node. On the other hand, other approaches are not control dependence conservative, like the ones mentioned for object-oriented programs ([16,17,24]) because they arbitrarily represent the unfolding of object variables with the use of control arcs not computed by Definitions 2 and 4.

The concept of structural dependence was already noted by previous authors along the literature. Krinke and Snelting [14], in their fine-grained program representation, decomposed program statements and noted a new kind of dependence they called *immediate control dependence* which, in fact, is similar to our defined structural dependence. Unfortunately, the representation of Krinke and Snelting cannot be applied to the SDG because of its granularity level. Their fine-grained PDG split every single statement in a set of nodes and this immediate control dependence was only usable for connecting the different parts of the statement. Additionally, the main purpose of the immediate control dependence during the slicing phase was to avoid the appearance of loops by limiting the slicing traversal. Also Cheda et al. [3], in their approach to static slicing for first-order functional programs which are represented by means of rewrite systems, designed the *term dependence graph* (TDG), that includes a set of arcs labelled with S for being considered *Structural* when representing the dependence between method calls and their corresponding arguments. Their TDG is only usable for first-order functional programs which are represented by means of rewrite systems and is not a derivation of the PDG. The idea of considering structural arcs to connect calls and arguments is promising if applied to SDG program slicing, but it only solves one of the incompatibilities detected in this paper.

7 Conclusions

In this paper we identify an important problem in program slicing: various standard solutions to different well-known slicing problems can be incompatible when they are combined in the same model. In particular, we have identified, with a set of associated counterexamples, different combinations of SDG extensions (for the treatment of exception-handling, unconditional jumps, and OO programs) that are incompatible, as they may produce incorrect slices.

The root of these problems is that those techniques extended the original SDG to solve different slicing problems in an independent way, without considering that their restrictions imposed over the slicing traversal could not be satisfied by the algorithms proposed in other solutions. In this paper we take a different perspective: we consider different problems all together and try to integrate the solutions proposed. From the best of our knowledge, there does not exist any theoretical work that integrates the techniques discussed in this paper.

After having identified and explained the rationale of the incompatibilities that make it impossible to combine those techniques, we have also proposed a solution for each incompatibility. The technical reason in all the cases is the interpretation of control dependence. Control dependence preservation is a key factor to make a SDG model compatible with other models. We have joined and generalized the restrictions over control flow and control dependence that must be taken into account to make all the models compatible. This generalization should be considered in future proposals.

From our theoretical setting, we have been able to implement the first program slicer with support for all the Java constructs mentioned along the paper: object-oriented features (classes, objects, interfaces, inheritance, and polymorphism), exception handling (*try-catch*, *throw*, *throws*), and unconditional jumps (*break*, *continue*, *return*...).

References

1. Allen, F.E.: Control flow analysis. SIGPLAN Not. **5**(7), 1–19 (1970)
2. Ball, T., Horwitz, S.: Slicing programs with arbitrary control-flow. In: Proceedings of the First International Workshop on Automated and Algorithmic Debugging. pp. 206–222. AADEBUG '93, Springer-Verlag, London, UK, UK (1993)
3. Cheda, D., Silva, J., Vidal, G.: Static slicing of rewrite systems. In: Proceedings of the 15th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2006). pp. 123–136. Elsevier ENTCS 177 (2007)
4. Cheng, J.: Slicing concurrent programs. In: Fritzson, P.A. (ed.) Automated and Algorithmic Debugging. pp. 223–240. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)
5. Cheng, J.: Dependence analysis of parallel and distributed programs and its applications. In: Proceedings. Advances in Parallel and Distributed Computing. pp. 370–377 (1997). https://doi.org/10.1109/APDC.1997.574057

6. DeMillo, R.A., Pan, H., Spafford, E.H.: Critical slicing for software fault localization. SIGSOFT Softw. Eng. Notes **21**(3), 121–134 (May 1996). https://doi.org/10.1145/226295.226310, http://doi.acm.org/10.1145/226295.226310
7. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems **9**(3), 319–349 (1987)
8. Galindo, C., Pérez, S., Silva, J.: Data dependencies in object-oriented programs. In: 11th Workshop on Tools for Automatic Program Analysis (2020)
9. Galindo, C., Pérez, S., Silva, J.: Program slicing with exception handling. In: 11th Workshop on Tools for Automatic Program Analysis (2020)
10. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation. pp. 35–46. PLDI '88, ACM, New York, NY, USA (1988). https://doi.org/10.1145/53990.53994, http://doi.acm.org/10.1145/53990.53994
11. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Transactions Programming Languages and Systems **12**(1), 26–60 (1990)
12. Kinloch, D.A., Munro, M.: Understanding c programs using the combined c graph representation. In: Proceedings 1994 International Conference on Software Maintenance. pp. 172–180 (1994)
13. Krinke, J.: Static slicing of threaded programs. SIGPLAN Not. **33**(7), 35–42 (Jul 1998). https://doi.org/10.1145/277633.277638, http://doi.acm.org/10.1145/277633.277638
14. Krinke, J., Snelting, G.: Validation of measurement software as an application of slicing and constraint solving. Information and Software Technology **40**(11), 661 – 675 (1998). https://doi.org/https://doi.org/10.1016/S0950-5849(98)00090-1, http://www.sciencedirect.com/science/article/pii/S0950584998000901
15. Kumar, S., Horwitz, S.: Better slicing of programs with jumps and switches. In: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002). Lecture Notes in Computer Science (LNCS), vol. 2306, pp. 96–112. Springer (2002)
16. Larsen, L., Harrold, M.J.: Slicing object-oriented software. In: Proceedings of the 18th international conference on Software engineering. pp. 495–505. ICSE '96, IEEE Computer Society, Washington, DC, USA (1996), http://dl.acm.org/citation.cfm?id=227726.227837
17. Liang, D., Harrold, M.J.: Slicing objects using system dependence graphs. In: Proceedings of the International Conference on Software Maintenance. pp. 358–367. ICSM '98, IEEE Computer Society, Washington, DC, USA (1998), http://dl.acm.org/citation.cfm?id=850947.853342
18. Majumdar, A., Drape, S.J., Thomborson, C.D.: Slicing obfuscations: Design, correctness, and evaluation. In: Proceedings of the 2007 ACM Workshop on Digital Rights Management. pp. 70–81. DRM '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1314276.1314290, http://doi.acm.org/10.1145/1314276.1314290
19. Ochoa, C., Silva, J., Vidal, G.: Lightweight program specialization via Dynamic Slicing. In: Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming. pp. 1–7. WCFLP '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1085099.1085101, http://doi.acm.org/10.1145/1085099.1085101

20. Silva, J.: A vocabulary of program slicing-based techniques. ACM Computing Surveys **44**(3) (June 2012)
21. Sintaha, M., Nashid, N., Mesbah, A.: Katana: Dual slicing-based context for learning bug fixes (2022)
22. Soremekun, E.O., Kirschner, L., Böhme, M., Zeller, A.: Locating faults with program slicing: an empirical analysis. Empirical Software Engineering **26**, 1–45 (2021)
23. Tip, F.: A survey of Program Slicing techniques. Journal of Programming Languages **3**(3), 121–189 (1995)
24. Walkinshaw, N., Roper, M., Wood, M.: The java system dependence graph. In: Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation. pp. 55–64 (2003)
25. Weiser, M.: Program Slicing. In: Proceedings of the 5th international conference on Software engineering (ICSE '81). pp. 439–449. IEEE Press, Piscataway, NJ, USA (1981)
26. Zhang, Z., Li, Y., Guo, Y., Chen, X., Liu, Y.: Dynamic slicing for deep neural networks. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 838–850. ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3368089.3409676, `https://doi.org/10.1145/3368089.3409676`
27. Zhao, J., Cheng, J., Ushijima, K.: Static slicing of concurrent object-oriented programs. In: Proceedings of 20th International Computer Software and Applications Conference: COMPSAC '96. pp. 312–320 (1996). https://doi.org/10.1109/CMPSAC.1996.544182