Exception-sensitive program slicing ☆

Carlos Galindo, Sergio Pérez, Josep Silva *

VRAIN, Universitat Politècnica de València, Camino de Vera s/n, 46022, Valencia, Spain

A B S T R A C T

Program slicing is a technique for program analysis and transformation with many different applications such as program debugging, program specialisation, and parallelisation. The *system dependence graph* (SDG), the most commonly used data structure for program slicing, has been extended in several ways to manage exception handling constructs. In this paper, however, we show that the presence of exception-handling constructs can make even the extended SDG produce incorrect and incomplete slices. To solve this situation, we survey the current state of the art and merge and extend different approaches (that treat *throws*, *try-catch*, etc.) to produce a version of the SDG that is able to manage all of them, that always produces complete slices, and that increases its precision keeping the same time complexity. An interesting side result is the discovering of a new kind of control dependence: *conditional control dependence*, which is needed to properly represent catch statements.

1. Introduction

Program slicing [15] is a technique for program analysis and transformation whose main objective is to extract a *slice* from a program: the set of statements that affect a specific set of variables v at a given program statement s, called a *slicing criterion* (denoted as $\langle s, v \rangle$). Program slicing has many practical applications such as debugging [3], program specialization [11], and software maintenance [6], among others. Modern program slicers include mechanisms to handle specific features of programming languages such as non-terminating programs [13], arbitrary control flow [2], or exception handling [1]. Traditionally, there are two main indicators used to measure the quality of a program slice: completeness and correctness. A program slice is considered to be *complete* when it contains all the statements that influence the value of the slicing criterion. A program slice is considered to be *correct* when all the statements included in the slice do influence the value of the slicing criterion.

In this work we focus on program slicing in presence of exception handling constructs. In particular, we show that the current approaches to account for exception handling can produce incomplete slices, and we propose a solution to this problem. The most extended approach in the area of exception-aware program slicing (and the basis used in most publications) is the one proposed by Allen and Horwitz [1], which in turn extended Sinha's proposal [14]. It supports

```
 1 public void f() {          public void f() {          public void f() {            1
 2   try {                      try {                      try {                       2
 3     g();                       g();                       g();                      3
 4   }catch (Exception e) {}    }                          }catch (Exception e) {}      4
 5   g();                       g();                       g();                        5
 6 }                          }                          }                            6
 7                                                                                    7
 8 public void g() {          public void g() {          public void g() {            8
 9   throw new Exception();     throw new Exception();     throw new Exception();      9
10 }                          }                          }                           10
```

| (a) Original program | (b) Allen and Horwitz's slice | (c) The correct slice |

Fig. 1. Java program that throws two exceptions but captures only the first one.

throw, *try*, *catch*, and *finally* instructions. Nevertheless, despite being valid for some combinations of the aforementioned instructions, it does not completely support all possible combinations, resulting in incomplete slices, as can be seen in Example 1.

Example 1 (*Incompleteness when slicing* try-catch *constructs in [1]*). Consider the Java program shown in Fig. 1a, in which method f is the entry-point. Two exceptions are thrown, one per call to g, but only the first one is caught. If we pick line 9 as the slicing criterion ($\langle 9, \emptyset \rangle$), then the slice should consist only of the statements that are needed to execute line 9 twice (i.e., the same number of times as in the original program). The slice produced by Allen and Horwitz can be seen in Fig. 1b. It removes the whole catch block, and thus it is incomplete, as line 9 will only execute once, and then the program will exit.

The source of this error is that in Allen and Horwitz's approach *catch* blocks are included only in a specific case: the slicing criterion is or requires a variable defined inside the *catch* block. This only happens when a statement of the *catch* block is included in the slice and, consequently, control dependences force the *catch* itself to be included too. Unfortunately, this is insufficient, since it does not capture the complex control dependences generated by *catch* blocks. This counterexample shows that even empty *catch* blocks may be necessary in the slice.

1.1. Contributions

In this paper we define an integral solution that merges together the current program slicing extensions used to slice programs with exception-handling. This is an important milestone in the area because there is not any survey that relates the work done so far (some approaches focused on *throw* statements, others on the *try-catch*, etc.). Therefore, our first contribution is an integrated explanation of the different graphs and approaches proposed so far. We describe how these graphs are constructed incorporating different ideas and approaches all together. Moreover, as a second contribution, we present a counterexample that shows different limitations of the current solution. In particular, we show that none of the previous approaches can properly treat *catch* statements, which sometimes leads to incorrect or incomplete results.

Our third and most important contribution is the definition of a new technique to solve the unveiled incompleteness problem. Our proposal includes in its basis the ideas presented in previous models, and augments them with the definition of a new ternary dependence called *conditional control dependence*, modelled with two new kind of arcs in the SDG. This new kind of control dependence arises when *try-catch* structures are used, and accurately model the control dependence relationships between *catch* statements and statements executed inside the *try* block and after the *try-catch* structure. Our solution has been proven complete for all possible *try-catch* scenarios resulting, to the best of our knowledge, into the most accurate static analysis model of *exception-handling* aware program slicing. Finally, our fourth contribution is the implementation of the first slicer able to properly treat *throws*, *try-catch*, and exception sources (both unconditional and conditional). This implementation has been released as free, and has been empirically evaluated with a series of real examples that are also described.

The rest of the paper is structured as follows: Section 2 recalls the background about program slicing with exception handling and introduces some preliminary definitions. Section 3 analyses the cases in which a *catch* statement could be included in a slice and defines conditional control dependence on that basis. Section 4 describes how to represent programs with exceptions with arcs representing the new dependence. Section 5 shows an algorithm to slice the new program representation. Section 6 describes the implementation and empirical evaluation. Section 8 presents the related work, and Section 9 summarizes our results.

2. Background

To keep the paper self-contained, we first define the base concepts of *slicing criterion* and *static backward slice*.

Definition 1 *(Slicing criterion).* Given a program P, a *slicing criterion* for P is a tuple $\langle s, v \rangle$ where $s \in P$ is a single statement and v is a subset of P's variables.

Prior to the definition of static backward slice, we need to define what a sequence of values of a slicing criterion is:

Definition 2 *(Sequence of values).* Let P be a program and $\langle s, v \rangle$ be a slicing criterion for P. $seq(P, s, v)$ is the *sequence of values* to which each variable in v is evaluated each time the execution of P passes through s.

Note that in Definition 1 the variables in v may not appear in s, or v may be empty. In the first case, the value of the variable is kept in the sequence, even though it is not used in the statement. In the latter, no variable is evaluated, so the sequence of values becomes a sequence of empty values, repeated the number of times that program P executes s.

Definition 3 *(Static backward slice).* Given a program P and a slicing criterion $SC = \langle s, v \rangle$, S is a *static backward slice* of P with respect to SC if S fulfills the following conditions:

- S is an executable program.
- $S \subseteq P$: S is the result of removing zero or more statements from P.
- For any possible input, $seq(P, s, v)$ is a prefix of $seq(S, s, v)$.

2.1. Program slicing based on dependence graphs

The computation of a slice from a given program has been traditionally performed as a graph-reachability problem using the *system dependence graph* (SDG). The SDG is constructed starting from the *control flow graph* (CFG). However, the presence of unconditional jumps in exception handling scenarios (`throw` statements) makes the traditional control flow graph (CFG) unsuitable to represent the flow of the program. For this reason, Ball and Horwitz [2] proposed the *augmented control flow graph* (ACFG), which is able to manage the presence of unconditional jumps. Then, Kumar and Horwitz [10] improved the definition of control dependence to increase precision, defining the *pseudo-predicate dependence graph* (PPDG). In [10] the SDG is built by using the ACFG as the starting point to construct a PPDG, and finally a SDG:

$$ACFG \rightarrow PPDG \rightarrow SDG$$

We describe the SDG generation by showing how each one of these graphs is built.

2.2. Augmented control flow graph (ACFG)

The CFG contains a node for each statement of the program and two special nodes *Entry* and *Exit* that respectively represent the start and end of the computation. An arc $(s_1 \rightarrow s_2)$ connects two statements s_1 and s_2 if there exists an execution in which s_2 is immediately executed after s_1.

The CFG is often used to define *control dependence*, whose standard definition is the following:

Definition 4 *(Control dependence).* Let G be a CFG. Let n and m be nodes in G. A node m post-dominates a node n in G if every directed path from n to the *Exit* node passes through m. Node m is *control dependent* on node n if and only if m post-dominates one but not all of n's CFG successors.

However, Horwitz et al. [2] noted that the above definition is not correct in presence of unconditional jumps. To solve the problem, they first identified the class of unconditional jump statements (*return*, *break*, *throw*...) and called them *pseudo-predicates*. A pseudo-predicate is a predicate where the true branch is the destination of the unconditional jump and the false branch (called non-executable branch) points to the statement that would execute if the statement failed to jump. Then, they redefined the CFG as a new graph, the *augmented control flow graph (ACFG)*, that takes into account the pseudo-predicates.

Definition 5 *(Augmented control flow graph (ACFG)).* Given a procedure P, which contains a list of statements $s = \{s_1, \ldots, s_n\}$, the *augmented control flow graph* of P is a directed graph $G = (N, A)$, where $N = s \cup \{Enter, Exit\}$ and A is a set of arcs of the form $(a, b) \mid a, b \in N$. Nodes may be either statements, predicates, pseudo-predicates, or exit nodes. Statements have one outgoing arc; predicates have two, labeled *true* and *false*; pseudo-predicates are like predicates, but their *false* arc is non-executable; and *Exit* has no outgoing arcs. Each arc represents that the pair of instructions it connects can execute sequentially in some execution of P. Non-executable arcs are the exception, as their name implies. The start and end of the procedure are represented with the *Enter* and *Exit* nodes.

The ACFG only has one source node, *Enter*, and one sink node, *Exit*. The *Enter* node should be able to reach all other nodes, and the *Exit* node should be reachable from all other nodes.

```
1   int f(int a, int b){
2     int sum = a + b;
3     int op = g(sum, b);
4     return op;
5   }
```

```
6   int g(int x, int y){
7     int i = 0;
8     while (i < x) {
9       if (i > y)
10        break;
11      y--;
12      i++;
13    }
14    return y;
15  }
```

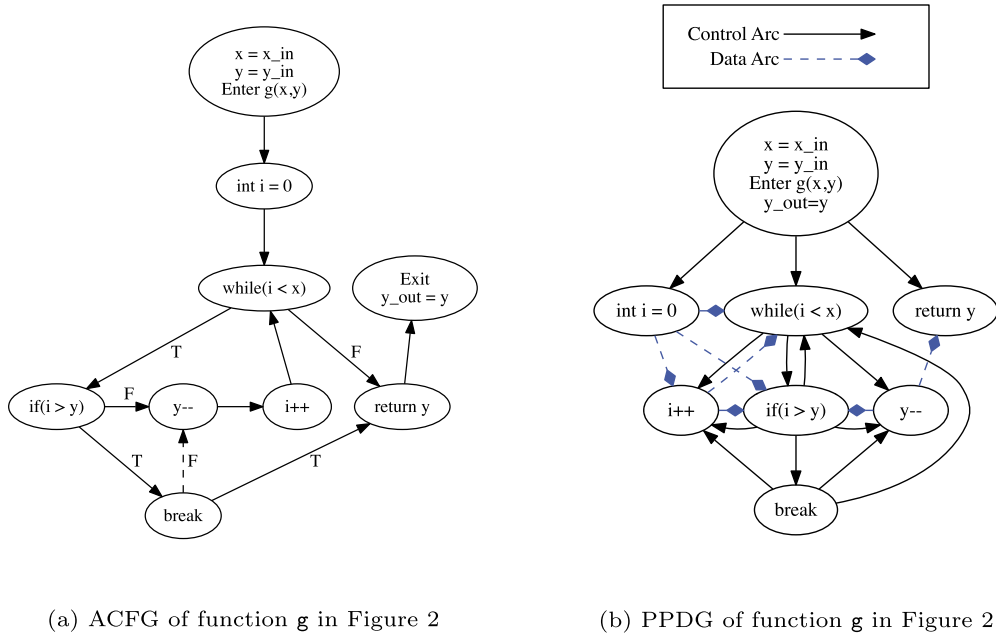**Fig. 2.** Fragment of code with two methods using call-by-reference.

(a) ACFG of function g in Figure 2

(b) PPDG of function g in Figure 2

Fig. 3. ACFG and PPDG of function g in Fig. 2.

Example 2. Consider the fragment of code in Fig. 2, which contains two methods f and g where f calls g. Note that method g contains an unconditional jump statement (break) in line 10.

Fig. 3a shows the ACFG representation of method g. The arcs of the ACFG are divided into executable arcs (solid) and non-executable arcs (dashed).

Note the representation of parameters: procedures with parameters, or that use or modify global variables have formal-in assignments in the *Enter* node ($a = a_{in}$), so that variables defined outside the procedure's body are defined in the graph; and formal-out assignments in the *Exit* node ($a_{out} = a$), so that changes made in the procedure can be passed back to the caller.

2.3. Pseudo-predicate program dependence graph (PPDG)

Once the ACFG has been built, two kinds of dependences are computed from it, which combined form the *pseudo-predicate program dependence graph* (PPDG) [10]. These dependences are data dependence and control dependence, which is often defined in terms of postdominance:

Definition 6 (*Postdominance* [4]). Let $G = (N, A)$ be an ACFG. $b \in N$ *postdominates* $a \in N$ if and only if b is present on every possible path in G from a to *Exit*.

Definition 7 (*Control dependence in the presence of pseudo-predicates* [10]). Let P be a procedure, let $G = (N, A)$ be its ACFG, and let $G' = (N, A')$ be its CFG. Given two nodes in the ACFG $a, b \in N$, b is *control dependent* on a if and only if b postdominates in G' one but not all of $\{n \mid (a, n) \in A, n \in N\}$ (a's successors in the ACFG).

Definition 8 *(Data dependence [10]).* Let $G = (N, A)$ be an ACFG. $b \in N$ is *data dependent* on $a \in N$ if and only if a defines a variable x, b uses x, and there exists in G a path free from non-executable arcs from a to b where x is not defined.

We can now formally define the PPDG.

Definition 9 *(Pseudo-predicate program dependence graph).* Given a procedure P and its associated ACFG $G = (N, A)$, the *pseudo-predicate program dependence graph* of P is a directed graph $G' = (N', A')$, where $N' = N \setminus \{Exit\}$ and $A' = A_c \cup A_d$, being A_c the set of control dependence arcs and A_d the set of data dependence arcs.

Example 3. Consider again the code in Fig. 2 and its ACFG shown in Fig. 3a. Its associated PPDG is shown in Fig. 3b. In this graph, in contrast to the PDG, the `break` node controls the execution of statements `y--` and `i++` (because of the non-executable arc of the ACFG) since the execution of the `break` prevents their execution.

It is important to note that, during the construction of the PPDG, the *Exit* node is removed, and the `formal-in` and `formal-out` assignments are all contained in the *Enter* node.

2.4. System dependence graph (SDG) and slice computation

After generating the PPDG, each `formal-in` and `formal-out` is split to its own node, control dependent on the *Enter* node, including a `formal-out` node for the result of the procedure. Each procedure call is unfolded into its own node, and nodes are generated to represent the input and output, called `actual-in` and `actual-out`. These are analogous to `formal-in` and `formal-out`. Then, the nodes structure formed in each procedure call is connected to the corresponding structure in the associated procedure definition through a new set of (interprocedural) arcs. These arcs are divided into input (from call to definition) and output (from definition to call) arcs and represent the information exchange between them. They connect all the PPDGs to form a single graph: the SDG. Finally, a new kind of arc called summary arc is added from `actual-in` nodes to `actual-out` nodes in procedure calls when needed. These arcs are added to procedure calls to illustrate data dependences inside the corresponding procedure definitions and are necessary to accurately slice procedure calls in the SDG with the algorithm proposed by Kumar and Horwitz in [10].

Once the SDG is built, the slicing algorithm can be used to compute the backward slice. First of all, we locate the node that corresponds to the slicing criterion and, starting from this node, all the arcs in the SDG are traversed backwards in two sequential phases. During the first phase the traversal ignores output arcs, and during the second one, it ignores input arcs. In any of the phases, when the traversal reaches a pseudo-predicate node different from the slicing criterion by traversing a control arc, the algorithm prevents the traversal to continue from this node. This process continues until no more arcs can be traversed. When the whole process finishes, the nodes reached by the algorithm form the so-called program slice.[1]

Example 4. Consider the SDG of the code in Fig. 2, represented in Fig. 4. The SDG represents both f and g. In the SDG, the PPDG of both method declarations is augmented with the addition of formal nodes for each variable used/defined inside the method. For example, in method g, the assignment nodes `x = x_in` and `y = y_in` represent `formal-in` nodes while the assignment node `y_out = y` represents a `formal-out` node. Analogously, method calls are augmented with the corresponding actual nodes. In this case, `x_in = sum` and `y_in = b` assignments in method call `g(sum,b)` represent the `actual-in` nodes while the `b = y_out` assignment represents the `actual-out` node of the call. Then, actual and formal nodes are linked with corresponding input/output arcs to represent parameter passing. Finally, summary arcs are computed for the call arguments and returned value. Once the SDG is built, the slicing algorithm defined in [10] can be now applied. Consider variable `op` in line 4 as the slicing criterion, where the node of the SDG representing its value is marked in bold. The corresponding program slice is marked in the SDG of Fig. 4 with grey nodes.

3. A new kind of dependence generated by catch statements

Example 1 reveals that the SDG proposed by Allen and Horwitz can generate incomplete slices because *catch* blocks are not correctly represented. In this section we explain the reason, showing that catch statements induce a kind of dependence that is not captured in any of the described graphs. Furthermore, we show that this kind of dependence cannot be captured by using traditional control dependence, and a new definition is needed.

A *catch* block is a statement that is only relevant if the program execution does not occur normally. For this reason, the control dependences they induce are slightly different from the ones generated by other statements. Instead of influencing other statements with their presence, it is their absence from the slice what may lead to a non-desired behaviour. We can illustrate this with the code in Fig. 5, which shows that the *catch* statement is not part of the slice even if the slicing criterion is a statement (line 3) inside the *try-catch* that throws the exception captured in the catch block (see Fig. 5b); or if

---

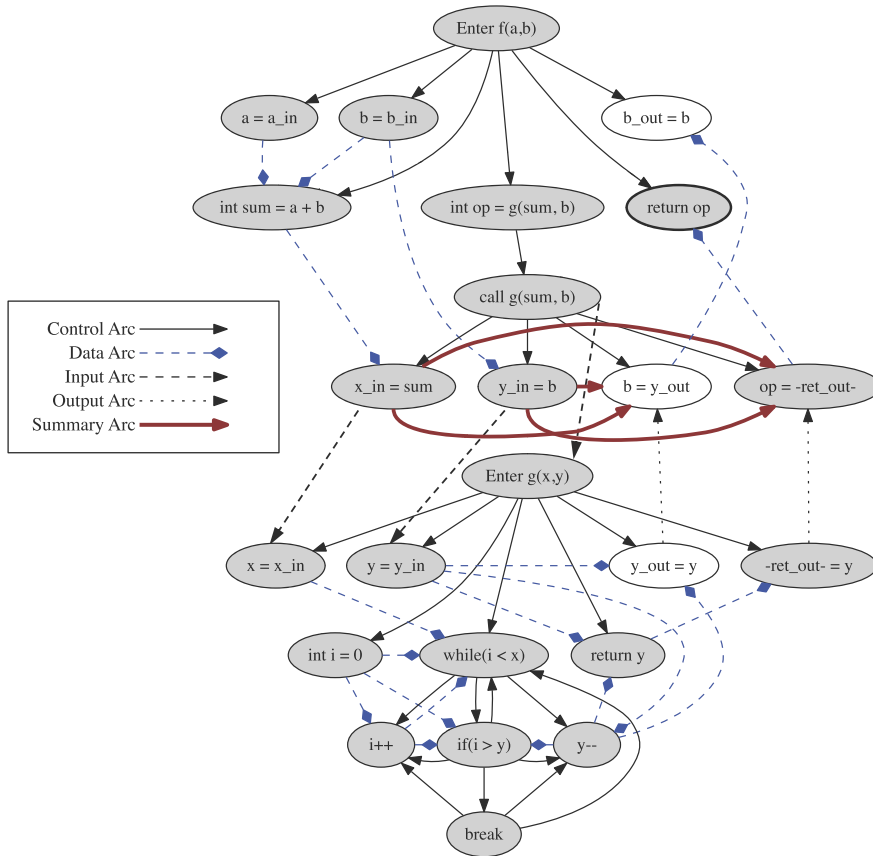[1] The interested reader can consult the paper by Kumar and Horwitz ([10]) for further explanations about the PPDG traversal.

Fig. 4. SDG of the code in Fig. 2 and slice with respect to ⟨4, {*op*}⟩.

(a) The original program.   (b) The slice with criterion ⟨3, ∅⟩.   (c) The slice with criterion ⟨5, ∅⟩.

Fig. 5. Slices of a code that throws and catches an exception.

the slicing criterion is located after the catch statement (see Fig. 5c). The *catch* statement should only belong to the slice if a statement that throws the captured exception belongs to the slice and also the *catch* statement affects some instruction that is also in the slice. These ideas are explained in the following three different slicing scenarios, which allow us to analyse how does the presence or absence of the *catch* statement in the slice affects other statements:

1. **Only the `throw` statement is part of the slice**. There is no reason for including the *catch* block in the slice if `log(x)` is not included in it. The slice would be lines 1, 3, and 6 (Fig. 5b).
2. **Only `log(x)` is part of the slice**. If only `log(x)` is in the slice, although the `catch` statement controls it, there is no possible statement inside the `try-catch` block in the slice to raise an exception that the `catch` captures, and thus the inclusion or exclusion of the `catch` statement does not influence the execution of `log(x)`. The slice would be lines 1, 5, and 6 (Fig. 5c).
3. **Both the `throw` statement and `log(x)` are part of the slice**. This situation is the counterpart of the previous one. In this case, `log(x)` is included in the slice, but there is also an exception source inside the `try` block that is part of the slice. Thus, to preserve the normal execution of the program and reach the `log(x)` statement, the `catch` block cannot be omitted. The slice would be the whole program (Fig. 5a).

These scenarios reveal the need for a new kind of control dependence that works in a conditional way. The catch instruction controls log(x) and throw new E() only if both of them are present in the slice (it controls both or none). This is because the catch instruction controls log(x) only if an exception that it can capture can be thrown, because the absence (rather than the presence) of the catch would change the number of times that log(x) is executed. Similarly, the catch instruction also controls the source of exceptions, but only when log(x) is included in the slice. This fact makes the control dependence of catch blocks completely different from any control dependence seen before. We call this new control dependence *conditional control dependence*.

Definition 10 (*Conditional control dependence*). Let $G = (N, A)$ be a CFG and $s_1, s_2, s_3 \in N$ be nodes in G. s_2, s_3 is *conditionally control dependent* on s_1 if s_1 is a catch statement, s_2 throws an exception that s_1 could capture, and s_3 is located outside s_1's body and there is a control-flow path from s_1 to s_3 in G.

4. Extending the SDG to make it exception-sensitive

In this section we introduce a procedure to build a SDG that contains conditional control dependences, so that *throw* statements, *try-catch* statements, exception sources (both conditional and unconditional), and procedures with exceptions can be properly represented and sliced. We present our solution as a set of modifications to the construction of the SDG described in Section 2. We organize our modifications considering the different graphs used to build a SDG: ACFG, PPDG, and finally SDG. In the following, to clearly differentiate between each version of the graph, our extended graphs are prefixed by 'ES-', which stands for "exception-sensitive", so the ACFG becomes the ES-ACFG, the PPDG becomes the ES-PPDG, and the SDG becomes the ES-SDG.

4.1. Modifications to the ACFG to create the ES-ACFG

In this section we compositionally describe how to construct any ES-ACFG: we show the graph representation of each syntax construct individually, but using a general representation that can be composed with the other constructs.

Like the ACFG, the ES-ACFG has three kinds of nodes: statements, which have only one outgoing arc; predicates, which have two outgoing arcs labeled *true* and *false* representing possible execution paths; and pseudo-predicates, which have two outgoing arcs labeled *true* and *false*, where the *false* arc represents a non-executable step.

Most instructions of the ACFG keep their traditional representation, but there are five constructs that need to be modified to properly account for exception handling: procedure declarations, procedure calls, and all those structures that cause or catch exceptions (e.g., *throw*, *try*, and *catch*). The rest of this subsection explains in detail these instructions and their correct representation.

Procedure declarations with exceptions. This case represents those function definitions that contain a potential source of exceptions, e.g., a *throw* statement, a possible division by zero, or a call to other procedure that may throw an exception. If the procedure contains exception sources, the *Exit* node contained in the original ACFG is split into three nodes: *normal exit*, *exception exit*, and *Exit* [1].

normal exit performs the function of the old *Exit* node, representing the exit from the procedure when no exception is raised. It is represented as a statement, whose arc is connected to *Exit*.

exception exit is the equivalent to *normal exit*, but it is only reached by nodes that generate uncaught exceptions. As it happened with *normal exit*, it is a statement whose arc is connected to *Exit*.

Exit is a sink node, to which the *normal exit* and *exception exit* nodes are connected.

Fig. 6 shows how this exit-node transformation is done showing the difference between an ACFG and an ES-ACFG when there is an exception source in a function definition. Additionally, as it can be seen, when there are formal-out associated to the function exit, they are moved to the specialised exit nodes, for increased precision. It is worth mentioning that nodes *exception source* and *exception exit* now include an assignment of the thrown exception (which we call "active exception", or ae for short) to propagate this exception until it is caught.

Calls to procedure definitions that may throw exceptions. These calls must be also redefined to differentiate whether the procedure ended with a normal execution (normal exit) or an exception was raised and uncaught during the execution (exception exit). The treatment is analogous to the one described for procedure definitions: with *normal return* and *exception return* nodes. The following changes are necessary:

The procedure call node is now a predicate, whose *true* arc is connected to *normal return* and the *false* arc, to *exception return*.

Normal return is a pseudo-predicate, whose *true* arc is connected to the following instruction, and its *false* arc is connected to the first instruction executed regardless of whether the *normal return* or *exception return* is executed. The destination of this *false* arc is necessary due to the new alternative execution path generated by the *exception return*. Adding the *false* arc to the first common instruction makes all the nodes after the call that are exclusively in the
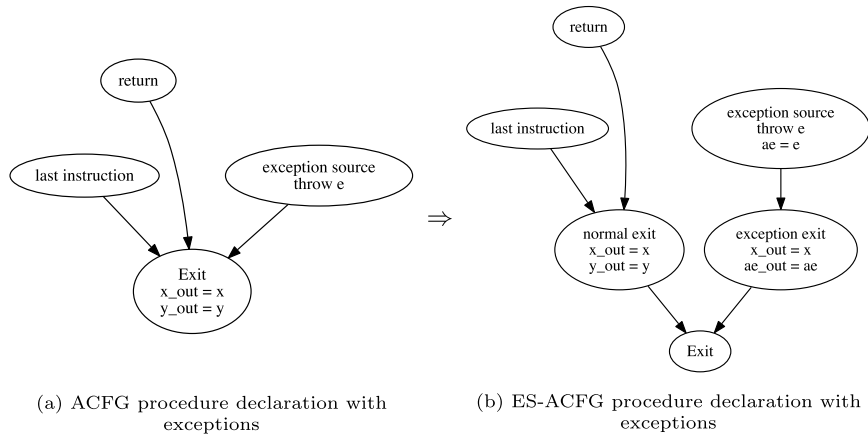
(a) ACFG procedure declaration with exceptions

(b) ES-ACFG procedure declaration with exceptions

Fig. 6. The ES-CFG uses two exit nodes (*normal exit* and *exception exit*) to differentiate normal and abrupt termination of a function.

(a) ACFG of procedure call with exceptions

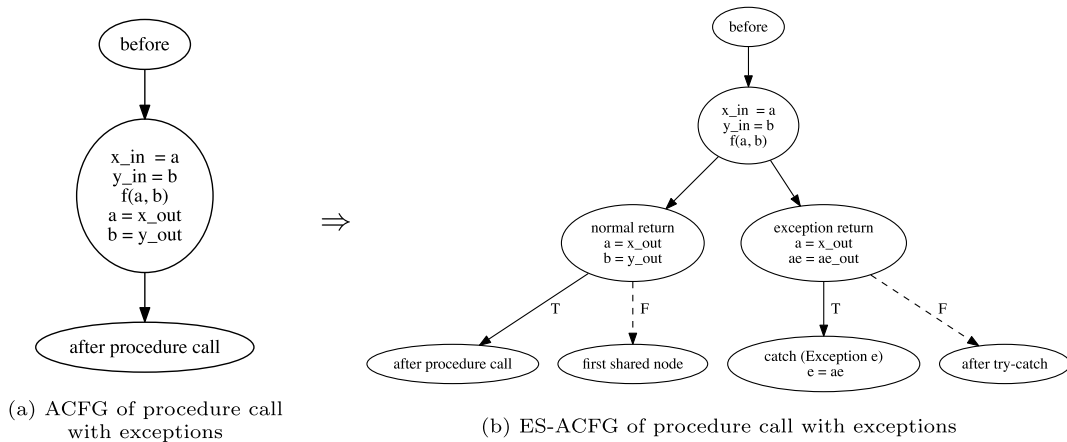(b) ES-ACFG of procedure call with exceptions

Fig. 7. The ES-CFG distinguishes between four paths to represent function calls that may produce exceptions.

normal execution path dependent on the normal return of the call, which is the expected semantic behaviour. Note that a common node between both paths always exists, and in the worst case scenario this node would be the *Exit* node.

Exception return is a pseudo-predicate, whose *true* arc is connected to the first *catch* node that may capture the thrown exception (or otherwise to the *exception exit* of the procedure), and its *false* arc is connected to the first node after the *try-catch* if it is contained in one, or otherwise to the *Exit* node.

The two *return* nodes contain assignments for modified global variables and parameters passed by reference. Fig. 7b shows the difference between the ACFG structure, where the behaviour is the same for both normal and exception procedure execution, and the ES-ACFG structure which differentiates both possibilities resulting into different execution paths. Note that both *return* nodes may not output the same variables, as some may have not been modified when the exception is thrown.

Unconditional exception sources. These instructions are those whose execution will always result on an exception being thrown or activated. They have an execution flow similar to the `return`, `break` or `continue` unconditional jump statements. For this reason, they are represented in the ES-ACFG as pseudo-predicates [1]. The *true* arc of the pseudo-predicate will be connected to the first *catch* instruction that can capture it, or, in case there is no *catch* able to capture it, to the *exception exit* node. The *false* arc will be connected to the instruction that would be executed if the pseudo-predicate failed to throw the exception, i.e., the next instruction in the sequential order of the source code. Fig. 8 shows an example of how a `throw` instruction is represented both in the ACFG and in the ES-ACFG.

Conditional exception sources. These instructions are the ones whose execution may activate an exception at runtime depending on the values given to variables during the execution, e.g., the operation `a = 10 / x` may generate a division by zero exception. This type of exception sources has the same representation as unconditional sources, but instead of being pseudo-predicates, they are predicates; to account for the fact that the exception really may or may not be thrown.
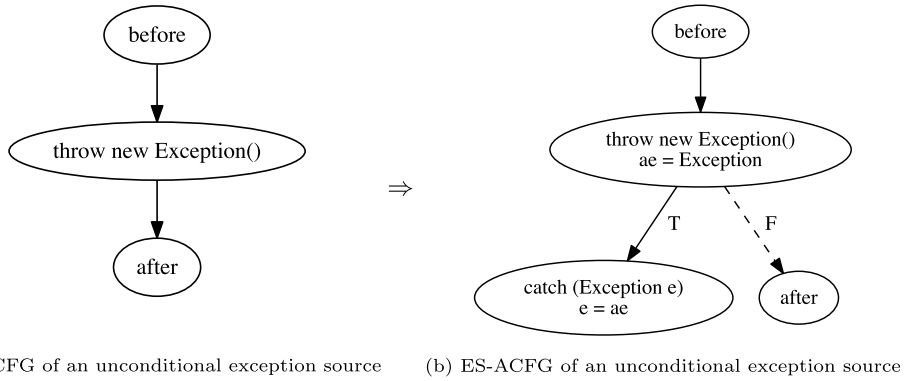
(a) ACFG of an unconditional exception source (b) ES-ACFG of an unconditional exception source

Fig. 8. Representation of a `throw` statement in the ACFG and ES-ACFG.

(a) ACFG of a conditional exception source (b) ES-ACFG of a conditional exception source
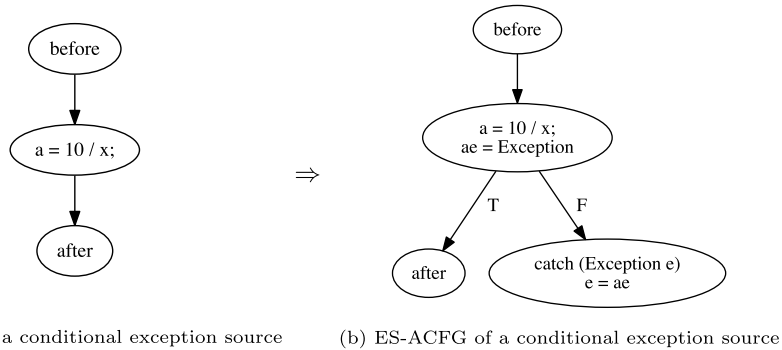
Fig. 9. Representation of a conditional exception source in the ACFG and ES-ACFG.

Fig. 9 shows an example, displaying the difference between the ACFG single path representation and the ES-ACFG representation, where two paths are now generated due to its predicate nature.

Exception catching structures. These structures are commonly called *try-catch* structures. In the original ACFG these structures were never considered, so we need to provide a representation that account for their control dependences correctly. The *try-catch* structure ES-ACFG representation is divided into its two different components:

try The *try* block represents the container of a sequence of statements where some of them may rise an exception. The way *try* blocks are represented is analogous to the representation of procedure definitions in [2]. They are considered pseudo-predicates, connecting their *true* arc to the first instruction within its body, and their *false* non-executable arc to the first instruction after the whole structure [1]. Thus, every instruction inside the *try* block is always controlled by the *try* itself. A scheme of how the *try* block is represented in the ES-ACFG is shown in Fig. 10a.

catch Each *catch* block is represented as a predicate or a pseudo-predicate depending on whether it captures or not all the exception sources connected to it. This means that the same *catch* block in different *try-catch* instructions can be represented in a different way. When all the exception sources connected to the *catch* block are captured by it, the block is represented as a pseudo-predicate, since the execution of the *false* arc (which let the execution flow continue to the exception exit) is non-executable (Fig. 10b). On the other hand, when any of the exception sources that reach the *catch* block may not be caught, the catch is represented as a predicate as both ES-ACFG paths can be executed at runtime (Fig. 10c).

4.2. Modifications to the PPDG to create the ES-PPDG

Once the ES-ACFG has been generated, the next step is to generate the corresponding PPDG by computing control and flow dependences. These dependences are computed in the same way as in [10] with a particular difference: while in the PDG construction shown in Section 2 `formal-out` assignments were moved to the *Enter* node after removing the *Exit* node, in the PPDG, `formal-out` assignments remain in the corresponding *normal exit* and *exception exit* nodes (see Fig. 6b). The resulting PPDG is a graph whose control dependences have slightly changed with respect to the original PPDG due to the changes introduced in the ES-ACFG. Although the new dependences provide new control dependences related to exception handling, the graph is not ready yet to deal with the conditional control dependence described in Section 3.
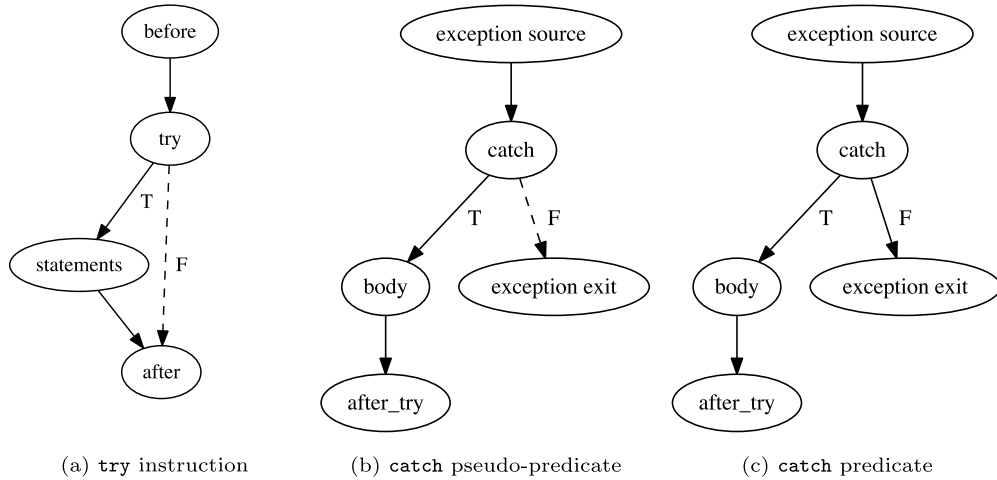
(a) `try` instruction   (b) `catch` pseudo-predicate   (c) `catch` predicate

Fig. 10. Representation of a `try-catch` statement in the ES-ACFG.

Algorithm 1 PPDG transformation.

Input: $G := (N, A)$, $A_c \in A$ (control dependence)
Output: $G' := (N, A')$
1: $A_{cc1} := \emptyset$, $A_{cc2} := \emptyset$, $A'_c := A_c$
2: **for all** $c \in N \mid$ IsCatch(c) **do**
3:    **for all** $(c, n) \in A_c \mid n \notin$ StmtsInBlock(c) **do**
4:       $A'_c := A'_c \setminus (c, n)$
5:       $A_{cc1} := A_{cc1} \cup (c, n)$
6:    **for all** $n \in$ TryStmtsOf(c) **do**
7:       **if** IsExceptionSource$(n) \wedge (n, c) \in A_c^*$ **then**
8:          **if** $\exists n' \mid (n, n') \in A_c^* \wedge (n', c) \in A_c^* \wedge n \neq n' \neq c$ **then**
9:             $A_{cc2} := A_{cc2} \cup (c, n)$
10: $A' := (A \setminus A_c) \cup A'_c \cup A_{cc1} \cup A_{cc2}$

Hence, a control dependence treatment needs to be done over the graph to classify and complement the control dependence arcs of the PPDG to obtain the ES-PPDG.

Algorithm 1 describes the process of adding conditional control dependence arcs to a PPDG. Each dependence generates two arcs, and they are placed in two sets: CC1 and CC2. This algorithm calls methods with descriptive names. For instance, function StmtsInBlock, that receives a *catch* node as its argument, returns a set with all the statements in its body; and TryStmtsOf(c) obtains the statements in the *try* block of the given *catch*. The operator $*$ in a set of arcs (e.g., A_c^*) represents its reflexive and transitive closure.

This algorithm analyses every *catch* node independently and divides its processing into two steps: the first (lines 2-5) selects every control arc from a *catch* node to a statement outside its body and converts it into a CC1 arc. The second step (lines 6-9) generates CC2 arcs from the *catch* node to each exception source in the *try*'s body, if there is a path of control arcs from the exception source to the *catch* node.

Note that conditional control dependence arcs (CC1 and CC2) are only created when the code contains at least one *catch* statement. In the case that no *catch* statement exists in the code, lines 2-9 cannot be executed, and sets A'_c, A_{cc1}, and A_{cc2} reach line 10 with their initial value given in line 1, thus, $A' = A$. Therefore, the PPDG and the ES-PPDG are equal when there are no *catch* statements.

4.3. From ES-PPDGs to the final ES-SDG

The creation of the ES-SDG can be described as the union of all the ES-PPDGs for each of the program's procedures, where the additional interprocedural and summary dependences are generated. The creation of input, output, and summary arcs is the same as in the SDG. The main difference between the standard SDG and the ES-SDG is the treatment of the different exit contexts. Every ES-PPDG may have either none or two *Exit* nodes: *normal exit* and *exception exit*. For this reason, the ES-SDG uses an output arc to connect an *exit* node in the declaration to its corresponding *return* node in the call. These can be seen in Fig. 11, where dotted arcs connect each *exit* to their corresponding *return* counterparts.
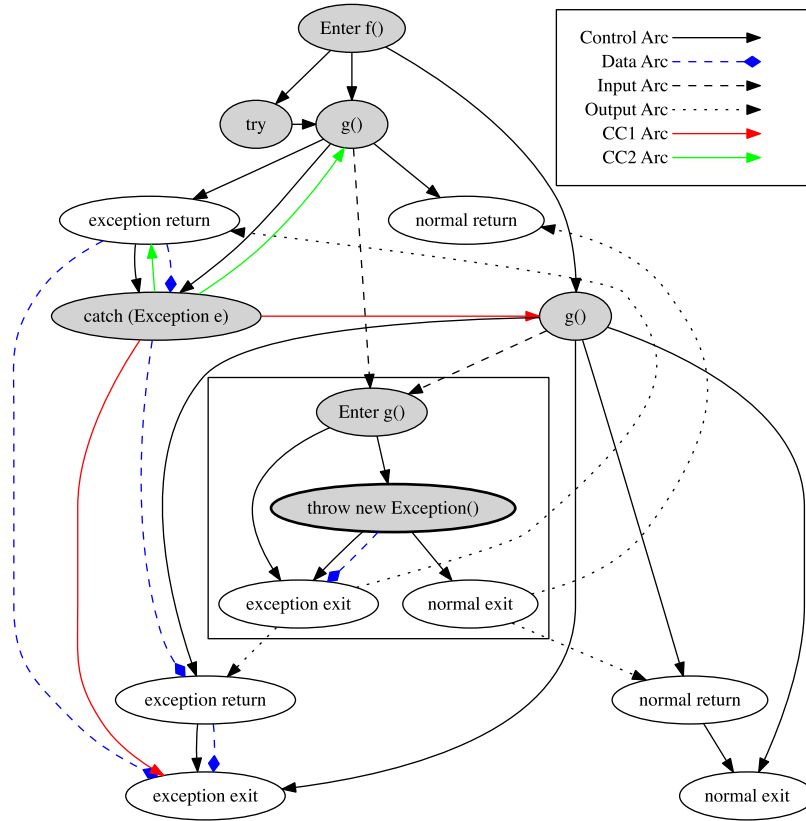
Fig. 11. The ES-SDG associated to the program in Example 1. The slicing criterion is represented with a bold node, and the central square separates the nodes that belong to g (inside) from the nodes that belong to f (outside). (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

5. Slicing conditional control dependence arcs

The ES-SDG introduces various structural changes and a new kind of arc: the conditional control dependence arcs. Therefore, the slicing algorithm must consider those changes. The new graph traversal is based on the slicing algorithm proposed by Horwitz et al. in [7], modified later by the introduction of the pseudo-predicates and the PPDG (see Algorithm 3 in [10]). In the ES-SDG, the presence of conditional control dependence arcs requires the introduction of some extra limitations on their traversal, to correctly represent its conditional nature:

1. If a node n is reached via a conditional control dependence arc of type t, it will not be included in the slice unless it has also been reached by another conditional control dependence arc of type t', such that $t \neq t'$. In that case, n's incoming arcs are not traversed, except if n is (also) reached during the slice traversal via another non-conditional arc (normal control, data, etc.).
2. Conditional arcs of type CC1 are transitive, even when the intermediate node is not included in the slice. For example, given $a \rightarrow^{CC1} b \rightarrow^{CC1} c$, if c is in the slice, a and b are both reachable via a conditional arc of type CC1, even when b is not in the slice. It is fundamental to mention that this transitive traversal is exclusively done at the end of each traversal phase and starts from any node in the slice. Each transitive traversal path ends when (i) it reaches a node that has only been reached by a CC2 arc (in this case, the node is included in the slice); or (ii) it reaches a node that was already included in the slice only by conditional arcs. This kind of transitivity is new in the SDG, and is required for cases where there is an exception source nested in more than one level of *try-catch* structures.

Algorithm 2 illustrates how restrictions 1 and 2 are included to the PPDG slicing algorithm described in [10]. Function SLICE represents the slicing process in two phases, function TRAVERSE performs the actual traversal of the graph in each phase, traversing the graph backwards and ignoring the set of arc types given in parameter *IgnoredTypes*, and function ADDTRANSITIVECC implements the transitive traversal of *CC1* arcs described in restriction 2. The algorithm defines sets *CC1R* and *CC2R*, which represent those nodes that have been reached by *CC1* and *CC2* arcs respectively. Additionally, another set *CCExclusive* is defined to store those nodes included in the slice exclusively by conditional control dependences. During the traversal, when we reach a node, the node is added to the set *PendingNodes*. The elements in *PendingNodes* are extracted one

by one during the traversal and the algorithm analyses them to apply restriction 1 when necessary. Then, all the incoming arcs of the extracted node are considered in the traversal. For each arc, if the *arcType* is contained in the list of *IgnoredTypes* (lines 12-13) or the pair $\{n, arcType\}$ does not respect the PPDG traversal restriction (lines 14-15) the arc is ignored and the traversal continues by extracting the source node of the arc m. In case the arc type is *CC1* or *CC2*, the node m is stored in the corresponding *CC1R* or *CC2R* sets respectively (lines 18-21). After adding the node to the corresponding set, the algorithm checks whether it is contained in both sets, adding the node to the slice (line 25). Additionally, if the node has not been reached and added to the slice before, it is also added to the *CCExclusive* set (lines 22-24). In case the arc type is not conditional control, m is included in the slice and in the *PendingNodes* set (lines 27-30). Moreover, if m was previously in the slice after being reached exclusively by arcs *CC1* and *CC2*, then it is removed from the *CCExclusive* set (line 30). Finally after traversing all possible arcs, the traversal tries to include transitive dependences of CC1 arcs by a call to procedure ADDTRANSITIVECC (line 31).

Function ADDTRANSITIVECC considers all the nodes reached by a *CC1* arc during the execution of the while loop in function TRAVERSE. For all these nodes, the function checks whether they have been included in the slice only by conditional control dependences, i.e., they are in the *CCExclusive* set (line 36). If they are not in the *CCExclusive* set, the incoming *CC1* arcs are traversed iteratively adding to the slice those reached node that have also been reached by *CC2* arcs (lines 37-42).

The complexity of the new traversal algorithm remains linear with respect to the number of nodes and arcs in the ES-SDG. This is because the changes to the algorithm are to stop the traversal when certain conditions are met; therefore lowering the amount of nodes reached. Additionally, each condition check can be made in constant time, and thus slicing remains linear. Example 5 shows the ES-SDG and the traversal of the slicing algorithm for the code of Example 1.

Example 5. If we apply Algorithm 2 to the problem shown in Example 1, we obtain the ES-SDG slice shown in Fig. 11. In this graph, the slicing criterion $\langle 9, \emptyset \rangle$ produces the slice composed of the grey nodes. The slice is computed as follows: First, the *Enter g()* node is included from the slicing criterion, which in turn includes both calls to procedure g. The first call causes the inclusion of the `try` and *Enter f()* nodes. Finally, thanks to the conditional arcs, the *catch* node is included, producing the expected slice in which the exceptions generated by g's first call may be caught and g's second call may be executed.

Algorithm 2 Slicing Algorithm for the ES-SDG.

Input: A ES-SDG G and the slicing criterion node n_{sc}.
Output: The set of nodes that compose the slice S of G w.r.t. n_{sc}.
Initialization: $CC1R := \emptyset, CC2R := \emptyset, CCExclusive := \emptyset$.

```
 1: function SLICE(G, n_sc)
 2:     S_0 := {n_sc}
 3:     S_1 := TRAVERSE(G, S_0, n_sc, {Output})
 4:     S := TRAVERSE(G, S_1, n_sc, {Input})
 5:     return S

 6: function TRAVERSE(G, N, n_sc, IgnoredTypes)
 7:     PendingNodes := N
 8:     while PendingNodes ≠ ∅ do
 9:         select some n ∈ PendingNodes
10:         PendingNodes := PendingNodes \ n
11:         for all arc ∈ GETINCOMINGARCS(n) do
12:             if arcType ∈ IgnoredTypes then
13:                 continue
14:             if isPseudoPredicate(n) ∧ n ≠ n_sc ∧ arcType = Control then
15:                 continue
16:             m := GETSOURCENODE(arc)
17:             if arcType = CC1 ∨ arcType = CC2 then
18:                 if arcType = CC1 then
19:                     CC1R := CC1R ∪ m
20:                 else
21:                     CC2R := CC2R ∪ m
22:                 if m ∈ CC1R ∧ m ∈ CC2R then
23:                     if m ∉ N then
24:                         CCExclusive := CCExclusive ∪ m
25:                     N := N ∪ m
26:             else
27:                 N := N ∪ m
28:                 PendingNodes := PendingNodes ∪ m
29:                 if m ∈ CCExclusive then
30:                     CCExclusive := CCExclusive \ m
31:         N := ADDTRANSITIVECC(G, N)
32:     return N
```

```
33: function ADDTRANSITIVECC(G, N)
34:     CC1Pending := CC1R
35:     for all n ∈ CC1Pending do
36:        if n ∉ CCExclusive then
37:           for all arc ∈ GETINCOMINGCC1ARCS(n) do
38:              m := GETSOURCENODE(arc)
39:              if m ∈ CC2R then
40:                 N := N ∪ m
41:                 continue
42:              CC1Pending := CC1Pending ∪ m
43:     return N
```

Table 1
Mean times required to generate and slice each SDG implementation.

Program	SC	G (AllenSDG)	G (ES-SDG)	S (AllenSDG)	S (ES-SDG)
B1.java	$\langle 28, \emptyset \rangle$	17.256 ms	18.977 ms	897 μs	1180 μs
B2.java	$\langle 14, \emptyset \rangle$	13.068 ms	13.266 ms	796 μs	1130 μs
B3.java	$\langle 28, \emptyset \rangle$	12.814 ms	13.297 ms	501 μs	798 μs
B4.java	$\langle 11, \emptyset \rangle$	4.795 ms	4.818 ms	144 μs	167 μs
B5.java	$\langle 18, \emptyset \rangle$	13.947 ms	14.133 ms	71 μs	73 μs
B6.java	$\langle 25, \emptyset \rangle$	13.565 ms	13.947 ms	912 μs	1219 μs
B7.java	$\langle 11, \emptyset \rangle$	3.095 ms	3.168 ms	105 μs	122 μs
B8.java	$\langle 18, \emptyset \rangle$	5.584 ms	5.598 ms	128 μs	200 μs
B9.java	$\langle 9, \emptyset \rangle$	2.629 ms	2.660 ms	111 μs	124 μs
Total		9.689 ms	9.940 ms	407 μs	557 μs

6. Empirical evaluation

In order to determine the degree to which the incompleteness of previous approaches has affected exception-handling constructs in program slicing, we have implemented the ES-SDG in a program slicer for Java. In order to perform a fair comparison, we implemented previous approaches to program slicing with exceptions. Our Java program slicer has been used as a baseline for both implementations, such that both have the same handling of objects, jumps, loops and other constructs unrelated to exception handling.

The slicer can be found in a public git repository,[2] and it contains two approaches to exception-sensitive programs slicing: Allen and Horwitz's [1] (`AllenSDG.java`) and our own (`ESSDG.java`). The whole implementation contains 11K lines of code, and it is available under a free software license.

We strictly followed the Georges et al.'s methodology [5]. We executed each graph generation and slice repeatedly. From each sequence of executions we extracted all the windows of 10 measurements where steady-state was reached, i.e., where the coefficient of variation (CoV, the standard deviation divided by the mean) of the 10 iterations is under 0.01. If no such window could be found, we selected the window of 10 measurements with the lowest CoV. The extracted windows were used to compute the average time to perform the given operation (build the graph or slice it).

The results of the experiments can be seen in Table 1, where each row shows a specific benchmark (file and slicing criterion). The first two columns (*Program* and *SC*) identify each benchmark by filename and slicing criterion. The following two columns (*G (AllenSDG)* and *G (ES-SDG)*) show the time required to generate the graph (in ms). Finally, the last two columns (*S (AllenSDG)* and *S (ES-SDG)*) display the time required to slice the graph (in μs).

The changes between approaches are not significant enough to alter the time required to generate or slice the graph significantly. Although the generation of the ES-SDG introduces a performance downgrade, the slowdown introduced is incidental. On the other hand, the main benefit of using the ES-SDG is achieved: the slices generated are now complete, but that comes at a slight increase in the time of slicing. The time required by the ES-SDG is 20 to 25% longer than AllenSDG. This is due to the additional conditions and the extra *catch* nodes reached. Overall, the cost is low enough, given that most applications of program slicing strictly require slices to be complete.

7. Completeness of the ES-SDG

Given a program P and a slicing criterion C, a *static backward slice* of P with respect to C must contain all statements in P that may influence (in some execution) C (a precise definition of slice can be found in Definition 3). The following theorem states that the slices produced by Algorithm 2 are always static backward slices, thus it is complete.

Theorem 1 (*Completeness*). *Let P be a program and $G = (N, A)$ its associated ES-CFG. Let node $n_{sc} \in N$ be the node associated with the slicing criterion $\langle s, v \rangle$. Slice(G, n_{sc}) is a static backward slice with respect to $\langle s, v \rangle$.*

The proof of this theorem requires to consider all possible combinations of slicing criterion point, execution path, and kind of exceptions raised (captured or not by catches). Therefore, it has been moved to Appendix A. This result is particularly relevant because it is the first proof of completeness for a variant of the SDG to treat exceptions. Previous approaches, like [1] or [8], based their completeness proofs in the fact that the PDG is automatically computed from the CFG and thus, slices computed over a PDG are complete. However, their models introduced modifications in the way *throw* and *catch* statements (and the impact of their presence) are represented in the CFG intra and interprocedurally, mainly based on a "reasonable" modification in line with previous models used to treat unconditional jumps [10]. The lack of completeness proofs in previous approaches has produced a chain of successive improvements to incrementally cover different cases of incorrectness or incompleteness. In addition to collecting some key ideas used in those previous works, we have defined a new type of dependence that is not considered in the original PDG, together with the introduction of some PDG edges (CC2) that are not generated by the classic control algorithms. For these two reasons, we consider a completeness proof to be mandatory. In fact, this proof states that our approach finally covers all cases (uncovered in previous approaches), thus ensuring completeness.

8. Related work

We have already explained in Section 2 the evolution of the SDG to treat exceptions with the definition of the ACFG and the PPDG. Here, we want to complement by commenting some approaches that have been a milestone in this area and that have inspired our work or are related to it. One of the most relevant initial approaches to exception-aware program slicing was Allen and Horwitz [1], which took advantage of the existing representation of unconditional jumps to represent exception-causing instructions, such as `throw`. Regarding exception-catching constructs, they simulated the real control flow and added non-executable control flow to generate the extra dependences they needed inside *try-catch* blocks. Unfortunately, they failed to account for the conditional nature of *catch* statements. They did not consider the possibility of an exception escaping from the *catch* block and, thus, they did not represent the control dependence between this kind of *catch* statements and the code placed immediately after them.

Later, Jiang et al. [8] described a solution for C++. *catch* nodes are represented similar to an *if-else* chain, each trying to capture the exception before deferring onto the next *catch* or propagating it to the calling method. They also were aware of the necessity of representing data dependences from procedure calls to *catch* nodes, but did not generalize that concept to all exception sources and usages. Other approaches include Prabhu et al. [12], which centered around the exception system of C++, and its specific quirks and design choices; and Jie et al. [9], which combined object orientation and exception handling. Jie et al. focused on the object-oriented side, rather than on the exception side, for which they used an approach similar to Jiang et al.'s or Allen and Horwitz's.

9. Conclusions

Program slicing is a powerful software analysis technique, powered by the system dependence graph, a directed graph that represents instructions and their dependences. In this paper, we introduce a new approach for program slicing with exception handling, merging the results of previous publications, extending those results, and creating a general algorithm that is valid for most programming languages with exception handling.

We have presented a counterexample to the current state of the art, which reveals a problem of incompleteness present in the literature; and we have proposed a solution, which we have proven complete. This solution also improves the precision of the slices by using a new notion of control dependence called *conditional control dependence*, which allows for the conditional inclusion of *catch* statements only when there is a statement that requires an exception to be caught, and at the same time, there exists a source of exceptions. Thus, we limit the inclusion of *try-catch* instructions and exception sources to the minimum necessary to generate complete slices.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Completeness proof of Theorem 1

This appendix has been included to ease the reviewers' reading but, due to its length, it will be published as a separate technical report. Appendix A includes the proof of Theorem 1.

Theorem 1 (*Completeness*). *Let P be a program and $G = (N, A)$ its associated ES-CFG. Let node $n_{sc} \in N$ be the node associated with the slicing criterion $\langle s, v \rangle$. $Slice(G, n_{sc})$ is a static backward slice with respect to $\langle s, v \rangle$.*

Proof. Assuming that the SDG is already capable of producing slices for programs that do not contain exception-handling constructs, we only need to prove that the additions made do not modify the behaviour regarding other instructions, and

```
1  void f() {
2    S1;
3    throw new Exception();
4  }
5
6  void g() {
7    S1;
8    try {
9      S2;
10     throw new Exception();
11   } catch (Exception e) {
12     S3;
13   }
14   S4;
15 }
```

```
16
17 Exception ex;
18 void h() {
19   S1;
20   try {
21     S2;
22     throw ex;
23   } catch (Exception e) {
24     S3;
25   }
26   S4;
27 }
```

Fig. A.12. Three procedures which throw an exception unconditionally, with no exception handling (f), complete exception handling (g), and partial exception handling (h).

that the behaviour related to exception-generating and handling constructs produces static backward slices (Definition 3). We prove the Theorem by induction on the size of the program.

Base case: We first consider the case when only one single `try-catch` block appears in the code, and make a case analysis to show that the slice produced is valid. For this, we study all possible places where the slicing criterion can be placed (inside the `try` block, at the `catch` instruction, inside the `catch` block, after the `catch` block...) and we also consider all possible situations that can happen depending on the exceptions raised (captured or not captured by each `catch`). This case is proved in Appendix A.1, where all possible combinations of a single exception source (either unconditional, conditional or a procedure call) with an exception-catching mechanism (or lack of it) are considered.

Induction hypothesis: We assume as the induction hypothesis that all the slices produced are valid (they fulfil the conditions in Definition 3) with a program with n nested `try-catch` blocks.

Inductive case: Finally, we prove the inductive case, when $n + 1$ `try-catch` blocks are nested in any of the possible combinations mentioned. This is proven in Appendix A.2 with another exhaustive case analysis for all cases where the inner `try-catch` could contain any number of `try-catch` structures. Each combination is composed of three parts: the original code, the ES-SDG, and all the possible slices.

As both the inductive and the base case are proven, we can assert that all combinations of exception-related instructions are handled properly by the ES-SDG, being Algorithm 2 complete: it produces static backward slices in all cases.

A.1. Exception sources and simple exception-catching structures

Throughout the rest of the proof, we introduce instructions before and after each exception causing or catching exceptions, and they are labeled S_n, where n is a unique identifier in that procedure. They affect neither control nor data flow, and their purpose is to display the effects of exception-related instructions in normal instructions.

There are three kinds of exception sources: conditional, unconditional and procedures through which exceptions propagate. On the other hand, we can consider three distinct cases: the exceptions generated are not caught (there is no `try-catch`), they are partially caught (the `try-catch` lets some through and catches some), or they are completely caught (the `try-catch` captures all of them). If we combine both, there are nine distinct possibilities to consider.

In each case, we consider and describe all possible slices given the following slicing criteria: all nodes are possible statements, but we will consider that the set of variables is always the empty set, as the problem of exception handling is orthogonal to data dependence: the only relevant variable is the active exception, but it cannot be selected as criterion, as it is not a variable defined in the program.

A.1.1. Unconditional exception source

In this section we study the different possibilities produced by a single unconditional exception source, e.g. a `throw` statement. As any code after an unconditional exception source is dead code, we will not place there a S_n instruction. Cases 1, 2, and 3 display the behaviour of unconditional exception sources.

Case 1 (*Unconditional exception source, exception not handled*). Consider procedure f, declared in lines 1-4 of Fig. A.12. It contains a single unconditional exception source, and no exception-catching instructions. Now consider its corresponding ES-SDG, shown in Fig. A.13. If the slicing criterion is either statement in the program (S1 or `throw`), only that statement and the *Enter* node is included in the slice. If the *exception exit* is reached via interprocedural arcs, S1 will not be included,
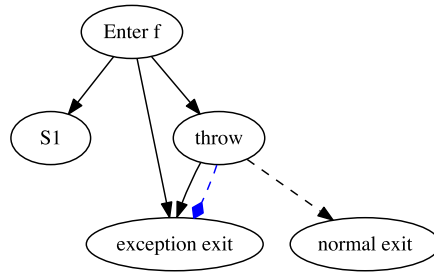
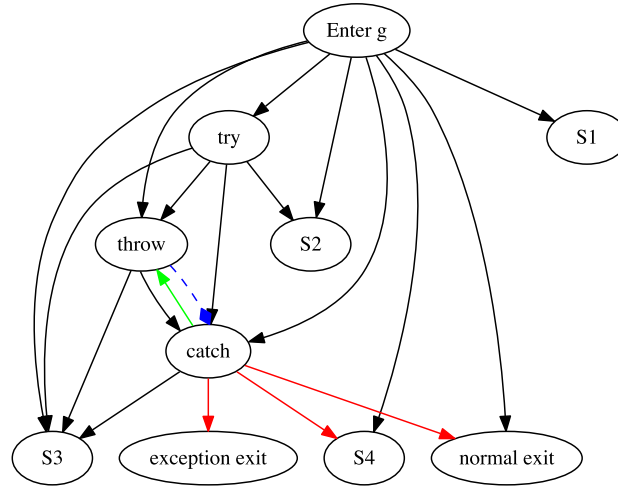**Fig. A.13.** ES-SDG corresponding to procedure f in Fig. A.12.

Fig. A.14. ES-SDG corresponding to procedure g in Fig. A.12.

as it is unnecessary for the slice. Finally, if *normal exit* is included in the slice, only the throw statement will be included. This can seem like an error in the ES-SDG, but *normal exit* is dead code (code that will never be executed), therefore all nodes that include (only) *normal return* and therefore *normal exit* are also dead code, and therefore the initial CFG is invalid.

Case 2 *(Unconditional exception source, exception completely caught).* Consider procedure g, declared in lines 6-15 of Fig. A.12. It contains a single unconditional exception source, and a try-catch instruction which catches all exceptions produced by the source. Now consider its corresponding ES-SDG, shown in Fig. A.14. Let's now consider which nodes should be included for each slicing criterion:

- S1 or S2: the S statement and the *Enter* node are included. In the second case, try is also included. This will require a post-processing to either extract S2 and remove try, or add a catch block; such that the slice is compilable.
- throw: because there is no instruction after the try-catch included in the slice, there is no need to include catch in it. As such, the slice consists of *Enter*, try and throw. A similar post-processing of the slice as in the previous item is needed to make the slice compilable.
- catch: in order to execute catch the same number of times, all exception sources should be included. And so it is, with the slice consisting of *Enter*, try, throw and catch. The S_n statements are unnecessary because the affect neither control nor data flow.
- S3: to execute an instruction in the catch's body, we need to include the catch itself, plus all its dependencies (see previous item). The resulting slice is the one in the previous item, plus S3.
- S4: because the catch captures all exceptions produced within try, the whole try-catch has no influence on instructions that come after it. Whether or not an exception is thrown, S4 will be executed. Therefore, no node affects S4, and the slice is *Enter* and S4.
- *Normal exit* behaves similarly to S4, because the only normal exit in the procedure comes immediately after it.
- *Exception exit* does not include any other node. This is because it is a "dead node", as no exceptions may escape the try-catch. However, if for other reasons the exception source is included, the catch node will be included via conditional arcs.
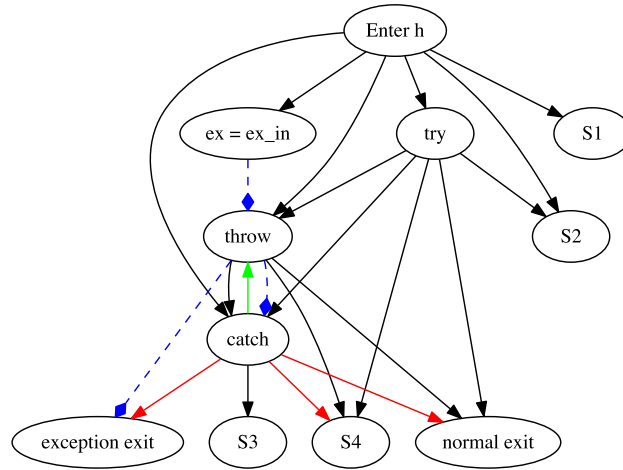
Fig. A.15. ES-SDG corresponding to procedure h in Fig. A.12.

```
1   void f() {
2     S1;
3     log(10 / 0);
4     S2;
5   }
6
7   void g() {
8     S1;
9     try {
10      S2;
11      log(10 / 0);
12      S3;
13    } catch (Exception e) {
14      S4;
15    }
```

```
16    S5;
17  }
18
19  void h() {
20    S1;
21    try {
22      S2;
23      log(10 / 0);
24      S3;
25    } catch (Exception e) { // TODO: not leaky!
26      S4;
27    }
28    S5;
29  }
```

Fig. A.16. Three procedures which throw an exception conditionally, with no exception handling (f), complete exception handling (g), and partial exception handling (h).

Case 3 (*Unconditional exception source, exception partially caught*). Consider procedure h, declared in lines 17-27 of Fig. A.12. It contains a single unconditional exception source, and a try-catch instruction which catches only some exceptions produced by the source. Now consider its corresponding ES-SDG, shown in Fig. A.15. Let's now consider the slices produced when each node is selected as the slicing criterion, considering that, except for S4, *normal exit* and *exception exit*; all other nodes have no extra incoming arcs with respect to 2, and therefore the slices are equal. The slices that change are as follows:

- S4: because the exception may not be caught, there are control dependencies from throw and try, which means that the inclusion of the catch is also necessary. In the end, the slice is the whole try-catch except for S2 and S3.
- *Normal exit*: exactly the same as S4, but with *normal exit* in the slice instead of it.
- *Exception exit*: compared to 2, it now has a data dependency from throw, which means that the whole try-catch (except for S2 and S3) is included in the slice.

A.1.2. Conditional exception source

In this section, we study the different possibilities produced by a single conditional exception source, e.g., a division where the divisor may be 0. The main two differences with unconditional exception sources are the fact that statements placed after it are no longer dead code and the change from pseudo-predicate to predicate, which lowers the number of control dependence arcs drawn (but not the actual dependences). As with unconditional exception sources, we place S_n-type instructions to analyse the behaviour at all possible points relative to the exception generation and capture. Cases 4, 5, and 6 display the behaviour of conditional exception sources.

Case 4 (*Conditional exception source, exception not handled*). Consider procedure f, declared in lines 1-5 on Fig. A.16. It contains a single conditional exception source, and no exception-capturing instructions. Now consider its corresponding ES-SDG, shown in Fig. A.17. Let us consider each node as the slicing criterion and see the resulting slice:
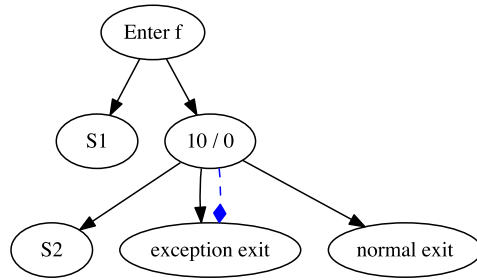
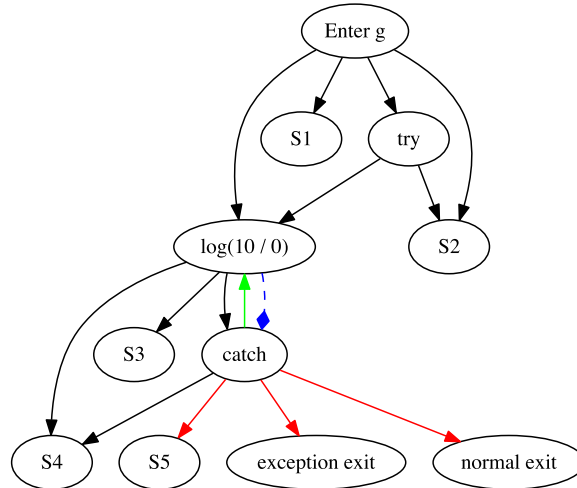**Fig. A.17.** ES-SDG corresponding to procedure `f` in Fig. A.16.

Fig. A.18. ES-SDG corresponding to procedure `g` in Fig. A.16.

- *Enter*: no other node is included.
- `S1` or `10/0`: only the slicing criterion and the *Enter* are included. In the case of `S1`, the exception source has no effect on it.
- `S2` or *exception exit*: *Enter* and the exception source are included in the slice, on top of the slicing criterion. The execution of the exception source is relevant to the execution or lack thereof of either slicing criterion, so it must be included.
- *Normal exit*: compared to 1, in which the exception source was unconditional, *normal exit* is no longer considered "dead code", and therefore is a valid slicing criterion which produces valid slices, which in this case includes *Enter*, the exception source and the slicing criterion.

Case 5 *(Conditional exception source, exception completely caught).* Consider procedure `g`, declared in lines 6-17 on Fig. A.16. It contains a single conditional exception source, which is captured every time by its surrounding `try-catch`. Now consider its corresponding ES-SDG, shown in Fig. A.18. When compared to unconditional exceptions, it is again very similar to the corresponding 2 and Fig. A.14: it has fewer arcs due to the exception source being a predicate and not a pseudo-predicate, and it has an additional node (`S3`). Notice how the addition has converted `S3` and `S4` in the unconditional case to `S4` and `S5` in the conditional case.

  Regarding the slices produced, they are the same for all nodes, except the newly introduced `S3`; whose slice would include *Enter*, `try`, the exception source and itself.

Case 6 *(Conditional exception source, exception partially caught).* Consider procedure `h`, declared in lines 19-29 on Fig. A.16. It contains a single conditional exception source, which is partially captured by its surrounding `try-catch`. Now consider its corresponding ES-SDG, shown in Fig. A.19. As with the previous example (5), there are many similarities with its unconditional exception counterpart (3): there are fewer control dependence arcs due to the conversion from pseudo-predicate to predicate, and there is an additional S_n node (`S3`). Again, the addition converts `S3` and `S4` to `S4` and `S5`.

  The slices produced by this graph are identical to those described in 3; and the slice of the newly introduced `S3`: *Enter*, `try`, the exception source and itself.
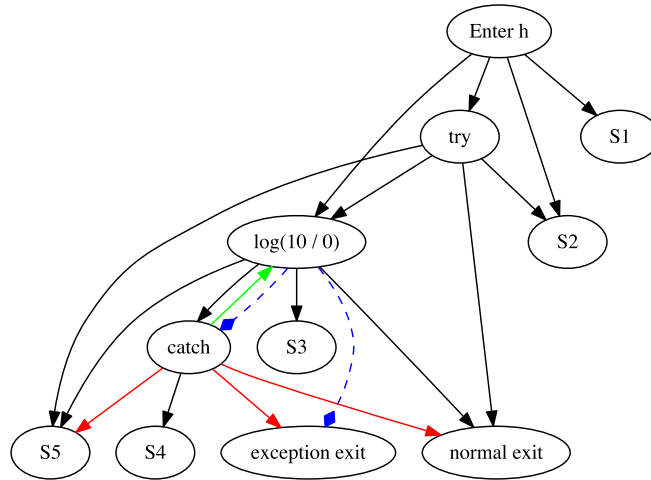
Fig. A.19. ES-SDG corresponding to procedure h in Fig. A.16.

```
1   void mayFail() {
2     if (cond)
3       throw new Exception();
4   }
5
6   void f() {
7     S1;
8     mayFail();
9     S2;
10  }
11
12  void g() {
13    S1;
14    try {
15      S2;
16      mayFail();
17      S3;
```

```
18    } catch (Exception e) {
19      S4;
20    }
21    S5;
22  }
23
24  void h() {
25    S1;
26    try {
27      S2;
28      mayFail();
29      S3;
30    } catch (IOException e) {
31      S4;
32    }
33    S5;
34  }
```

Fig. A.20. Three procedures in which a procedure that may throw an exception is called, with no exception handling (f), complete exception handling (g), and partial exception handling (h). The called procedure's code is displayed at the top (mayFail).

A.1.3. Procedures that throw exceptions

In this section, we study the different possibilities produced by a procedure call that may or not produce an exception. The handling is more complex, as the presence of *exception return* and *normal return* generate more variety. In spite of this, the dependencies generated are generally the same: the structures where exception sources are needed are the same; the representation of the exception is more granular. As with previous sections, we place S_n-type instructions, which behave in the control flow graph like statements, in order to be able to simulate and select all possible slicing criteria w.r.t. a procedure that may produce exceptions. Cases 7, 8, and 9 display the behaviour of procedure calls that may throw exceptions.

Case 7 (*Procedure that may throw exceptions, exception not handled*). Consider procedure f, declared in lines 6-10 on Fig. A.20. It contains a call to a procedure that may produce exceptions, mayFail, which in turn is declared in lines 1-4 on the aforementioned figure. Now consider its corresponding ES-SDG, shown in Fig. A.21. The call arc is represented with a dashed edge, and the return arcs are represented with dotted edges. Here are all nodes and the slices produced if they were selected as the slicing criterion:

- *Enter*: the resulting slice contains only itself.
- S1 or mayFail(): the resulting slice contains *Enter* and the slicing criterion.
- *Normal return* or *exception return*: the slice contains the corresponding *exit* node from the mayFail procedure definition, throw, if, *Enter mayFail*, mayFail(), *Enter* and the slicing criterion.
- *Exception exit*: the slice contains the same nodes as *exception return*'s slice, plus the *exception exit* itself.
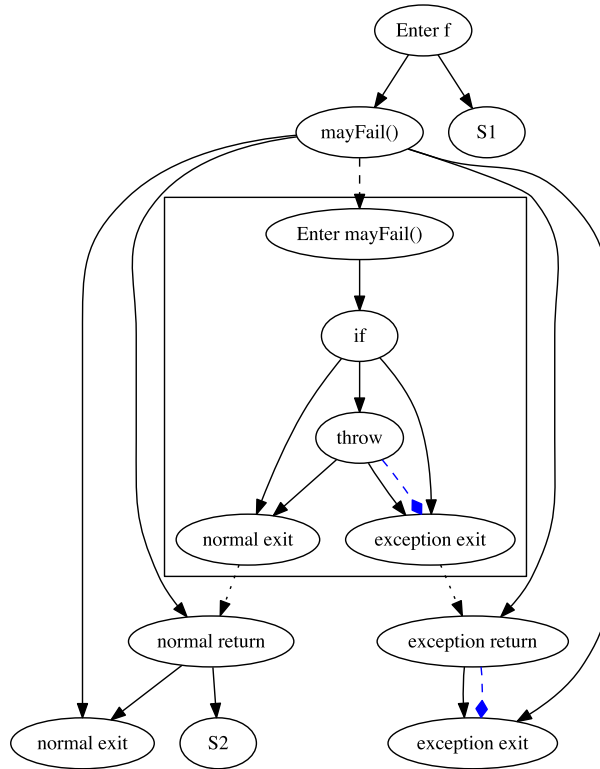- *Normal exit* or S2: the slice contains the same nodes as *normal return*'s slice, plus the slicing criterion.

Fig. A.21. ES-SDG corresponding to procedure f in Fig. A.20.

Case 8 *(Procedure that may throw exceptions, exceptions completely caught).* Consider procedure g, declared in lines 12-22 on Fig. A.20. It contains a call to a procedure that may produce exceptions, mayFail, which in turn is declared in lines 1-4 on the aforementioned figure. Now consider its corresponding ES-SDG, shown in Fig. A.22. The call arc is represented with a dashed edge, and the return arcs are represented with dotted edges. Here are all nodes and the slices produced if they were selected as the slicing criterion:

- *Enter*: the resulting slice contains only itself.
- S1 or try: the slice contains the slicing criterion and *Enter*.
- S2 or mayFail(): the slice contains the slicing criterion, *Enter* and try.
- Normal return or exception return: the slice contains the corresponding *exit* node from the mayFail procedure definition, throw, if, *Enter mayFail*, mayFail(), *Enter* and the slicing criterion.
- S3: the slice contains the nodes of *normal return*'s slice and the slicing criterion itself.
- catch: the slice contains the nodes of *exception return*'s slice and the slicing criterion.
- S4: the slice contains the nodes of catch's slice and the slicing criterion.
- S5 or *normal exit*: the slice contains the slicing criterion and *Enter*. It does not need any node from the try-catch, as all exceptions produced are captured. If for any reason an exception source (either mayFail() or *exception return*) is included, then the catch node would also be included, by virtue of the conditional control flow.
- *Exception exit*: no node is included in the slice, because there is no exception may reach this node. The only case where additional nodes will be included is when an exception source is present, and therefore the catch node would be needed.

Case 9 *(Procedure that may throw exceptions, exceptions partially caught).* Consider procedure h, declared in lines 24-34 on Fig. A.20. It contains a call to a procedure that may produce exceptions, mayFail, which in turn is declared in lines 1-4 on the aforementioned figure. Now consider its corresponding ES-SDG, shown in Fig. A.23. The call arc is represented with a dashed edge, and the return arcs are represented with dotted edges. Here are all nodes and the slices produced if they were selected as the slicing criterion:

- S5, *exception exit* or *normal exit*: the slice contains *Enter*, try, mayFail(), the complete procedure declaration of may-Fail, *normal return*, *exception return*, catch and the slicing criterion. In the case of *exception exit*, notice how the data
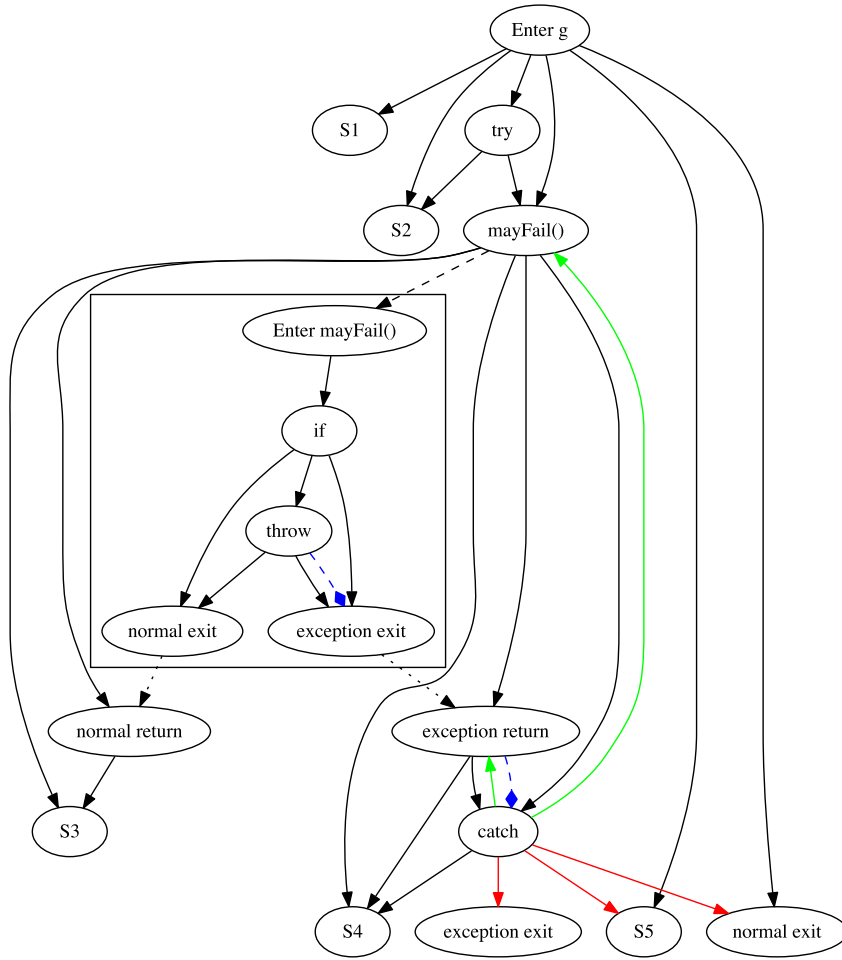
Fig. A.22. ES-SDG corresponding to procedure g in Fig. A.20.

dependency of the "active exception" picks up the *exception return* node, and otherwise the slice would include nothing more than *exception exit*.
• All other nodes behave in the same way as in 8.

A.2. Nested exception-catching structures

Consider a procedure with n `try-catch` instructions, all of them nested; and assume that the ES-SDG is capable of producing valid slices. Now consider the case where the outermost `try-catch` is surrounded by another `try-catch`, creating a nested structure of $n+1$ `try-catch` instructions. Each `try-catch` may contain other additional instructions and exception sources apart from the `try-catch` it holds, but it is not required to do so. In this section, we showcase the six possible combinations that the $n+1$st `try-catch` introduces in the system.

Throughout this section, we label all exception sources from within the n nested `try-catch` as the *inner* exception sources, and all exception sources from within the additional `try-catch` (but not within the n inner blocks) as the `outer` exception sources. Then, we consider where are these two kinds of exception sources captured. In the case of inner exceptions, it can either be in any of the inner `catch` blocks, in the outer `catch` block or nowhere in the procedure– meaning they propagate through the call stack. In the case of outer exceptions, due to them being outside the inner `try-catch` blocks, they cannot be captured by inner `catch` blocks, so they can be captured by either the outer `catch` block or nowhere in the procedure. Table A.2 shows the combination of the two locations where inner and outer exceptions are captured, which results in six different situations: six different ES-SDGs.

In each ES-SDG present, the code is the same, but the exceptions caught at each level vary. A simplified pseudocode is used, in order to avoid changing the `catch` and exception source's type to reflect where each exception is captured. The pseudo-code for the procedure can be seen in Fig. A.24, where instructions labeled S_n are statements without any effect on control or data dependence; exception sources are displayed as `inner_source` and `outer_source`; and catch
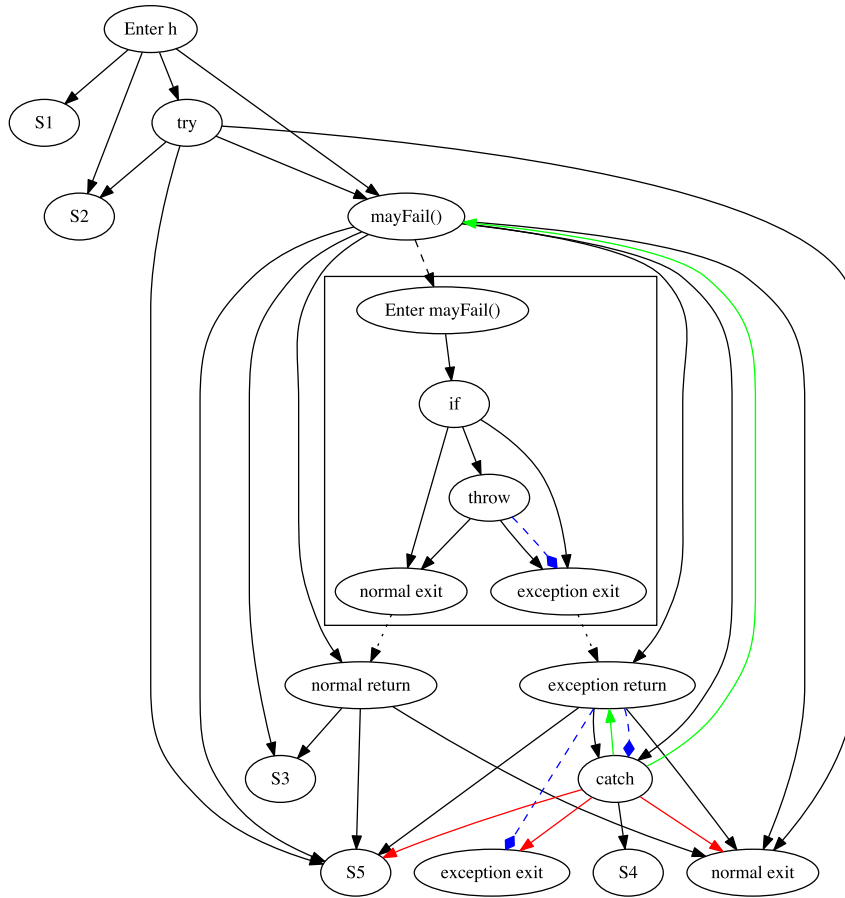
Fig. A.23. ES-SDG corresponding to procedure h in Fig. A.20.

Table A.2
All possible combinations for the location where inner and outer exception are caught (either in *inner* catch blocks, *outer* catch nodes or *none*).

Case number	1	2	3	4	5	6
Where are inner exceptions caught	inner	outer	none	inner	outer	none
Where are outer exceptions caught	outer	outer	outer	none	none	none

```
1   void nested() {
2     S1;
3     try {
4       S2;
5       try {
6         S3;
7         inner_source;
8         S4;
9       } catch (Inner i) {
10        S5;
11      }
12      S6;
13      outer_source;
14      S7;
15    } catch (Outer o) {
16      S8;
17    }
18    S9;
19  }
```

Fig. A.24. A procedure with two exception sources in two nested try-catch blocks. Instructions of the form S_n are statements that do not generate data or control dependencies.
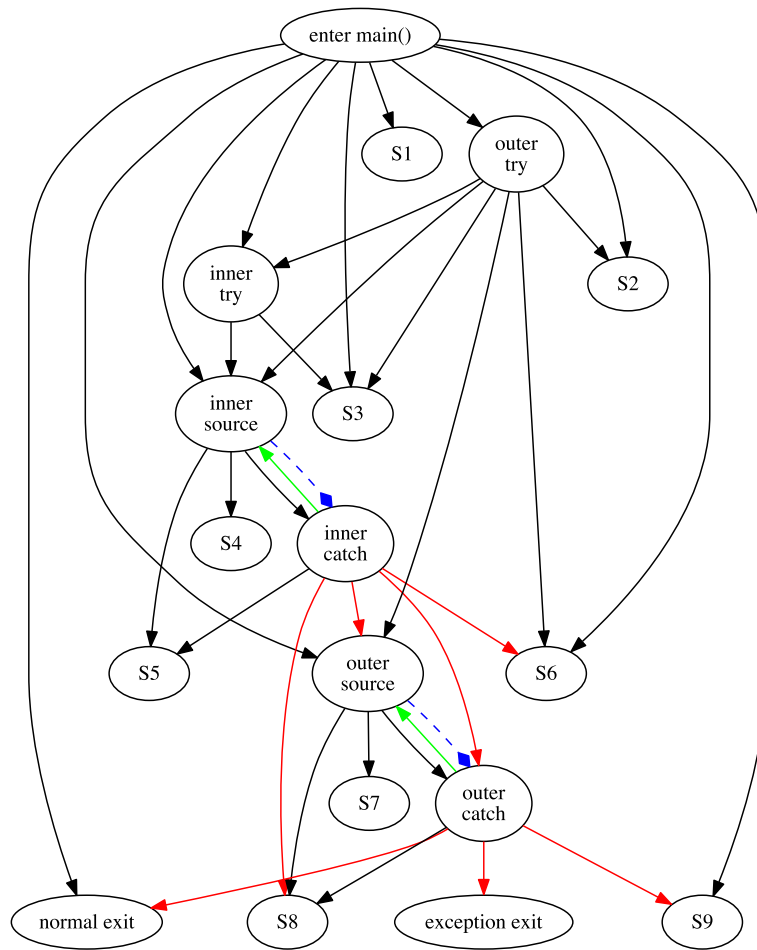
Fig. A.25. The ES-SDG corresponding to the code in Fig. A.24, in the case 1 of Table A.2.

instructions are labeled `Inner` and `Outer` to be distinguishable. The following sections showcase the ES-SDG produced for each situation and the slices that it results in.

A.2.1. Case 1

Consider the case when the exceptions produced in each source are contained at the same level; inner exception sources in the inner `catch` and outer exception sources in the outer `catch`. The corresponding ES-SDG for this case is shown in Fig. A.25. The slices produced by selecting each node are the following:

Enter Only the slicing criterion is in the slice.
S1, `try` (outer), S9 or *normal exit* The slice consists of the *Enter* node and the slicing criterion.
S2, `try` (inner), `outer_source` or S6 The slice consists of the *Enter* node, the slicing criterion and the outer `try` node.
S3 or `inner_source` The slice consists of the *Enter* node, the slicing criterion and both `try` nodes (inner and outer).
S4 or `catch` (inner) The slice consists of `inner_source`'s slice, plus the slicing criterion.
S5 The slice consists of inner `catch`'s slice, plus the slicing criterion.
S7 or `catch` (outer) The slice consists of `outer_source`'s slice, plus the slicing criterion.
S8 The slice consists of outer `catch`'s slice, plus the slicing criterion.
exception exit The slice consists of the slicing criterion, as no exception can reach that node and therefore, it is a "dead node".

It is also interesting to consider the case where, instead of selecting a slicing criterion, we select an initial set of nodes in the slice and continue from there. The first we could build would be to select both exception sources simultaneously. The result is that the inner `catch` is included, but not the outer one, as there are no instructions after it that need to be executed. Another one is selecting a statement after the `try-catch` (S9) and one of the exception sources. The resulting slice in this case would include the corresponding `catch` (the inner one for the inner source and the outer one for the
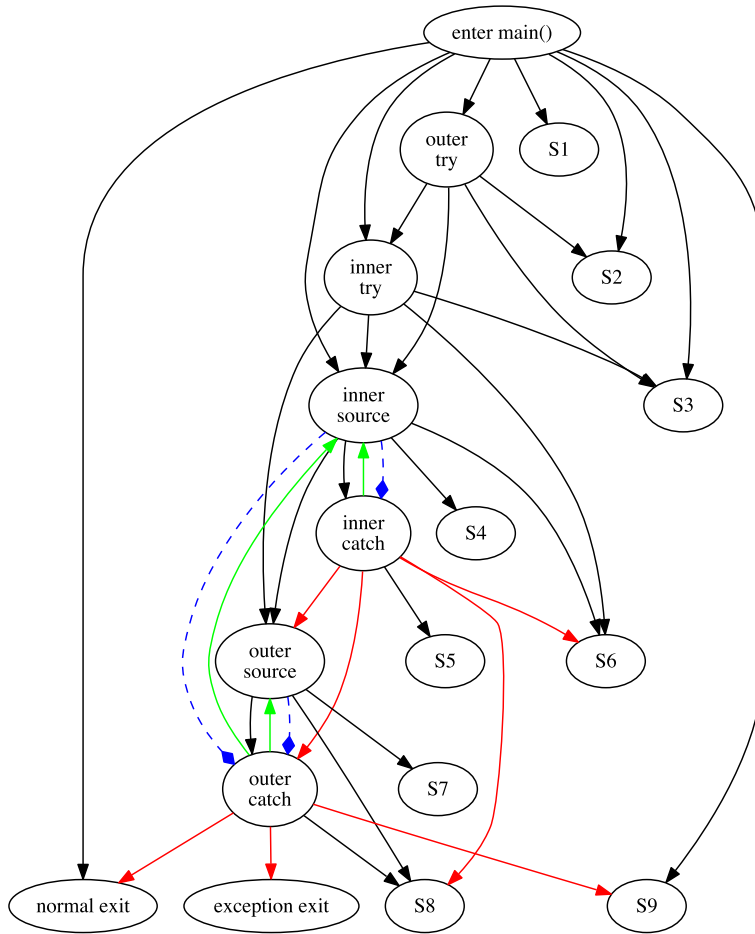
Fig. A.26. The ES-SDG corresponding to the code in Fig. A.24, in the case 2 of Table A.2.

outer source). Finally, if both exception sources are included, plus `S9`, both `catch` statements are necessary; and both are included in the slice via conditional arcs.

A.2.2. Case 2

Consider the case when the exceptions produced in the inner source are captured either in the inner or outer `catch`; and the exceptions produced in the outer source are captured in the outer `catch`. The corresponding ES-SDG for this case is shown in Fig. A.26. The slices produced by selecting each node are the following:

Enter Only the slicing criterion is in the slice.
`S1`, `try` (outer), `S9` or *normal exit* The slice consists of the *Enter* node and the slicing criterion.
`S2` or `try` (inner) The slice consists of the *Enter* node, the slicing criterion and the outer `try` node.
`S3` or `inner_source` The slice consists of the *Enter* node, the slicing criterion and both `try` nodes (inner and outer).
`S4`, `catch` (inner), `outer_source` or `S6` The slice consists of `inner_source`'s slice, plus the slicing criterion.
`S5` The slice consists of inner `catch`'s slice, plus the slicing criterion.
`S7` or `catch` (outer) The slice consists of `outer_source`'s slice, plus the slicing criterion.
`S8` The slice consists of outer `catch`'s slice, plus the slicing criterion.
exception exit The slice consists of the slicing criterion, as no exception can reach that node and therefore, it is a "dead node".

Notice how the control dependency arcs reflect the fact that `inner_source`'s exception is not completely captured by the inner `catch`, and therefore, the instructions that follow it are dependent on `inner_source`'s execution. In the case of `S4`, `S6` and `outer_source`, the control dependence from `inner_source` and the conditional arcs are the reason for the inclusion of `catch`.
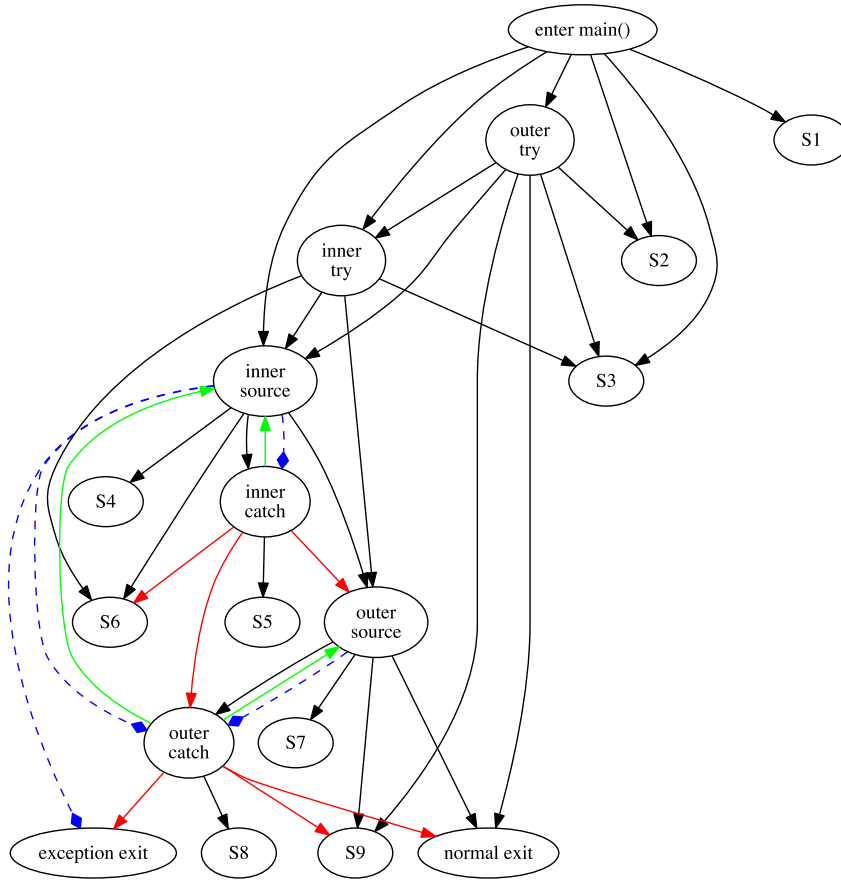
Fig. A.27. The ES-SDG corresponding to the code in Fig. A.24, in the case 3 of Table A.2.

A.2.3. Case 3

Consider the case when the exceptions produced in the inner source are partially captured either in the inner or outer `catch`; and the exceptions produced in the outer source are captured in the outer `catch`. The corresponding ES-SDG for this case is shown in Fig. A.27. The slices produced by selecting each node are the following:

Enter Only the slicing criterion is in the slice.
S1 or `try` (outer) The slice consists of the *Enter* node and the slicing criterion.
S2 or `try` (inner) The slice consists of the *Enter* node, the slicing criterion and the outer `try` node.
S3 or `inner_source` The slice consists of the *Enter* node, the slicing criterion and both `try` nodes (inner and outer).
S4, `catch` (inner), `outer_source` or S6 The slice consists of `inner_source`'s slice, plus the slicing criterion.
S5 The slice consists of inner `catch`'s slice, plus the slicing criterion.
S7, `catch` (outer), S9 or *normal exit* The slice consists of `outer_source`'s slice, plus the slicing criterion.
S8 The slice consists of outer `catch`'s slice, plus the slicing criterion.
exception exit The slice consists of *exception exit*, both `catch` nodes, `inner_source`, both `try` nodes and *Enter*.

In the case of *exception exit*, notice how (1) the inclusion of `catch` nodes is via conditional arcs, (2) the inclusion of the inner `catch` is performed thanks to the transitivity of conditional arcs, and (3) the `outer_source` is not included, as it is completely captured by the outer `catch`, which means that it cannot produce an exception that reaches *exception exit*. The same exercise of selecting multiple nodes can be performed, but most of them pick `inner_source` almost immediately, due to its exceptions never being completely captured.

A.2.4. Case 4

Consider the case when the exceptions produced in the inner source are captured in the inner `catch`, and the exceptions produced in the outer source are not completely captured. The corresponding ES-SDG for this case is shown in Fig. A.28. The slices produced by selecting each node as the slicing criterion are the following:
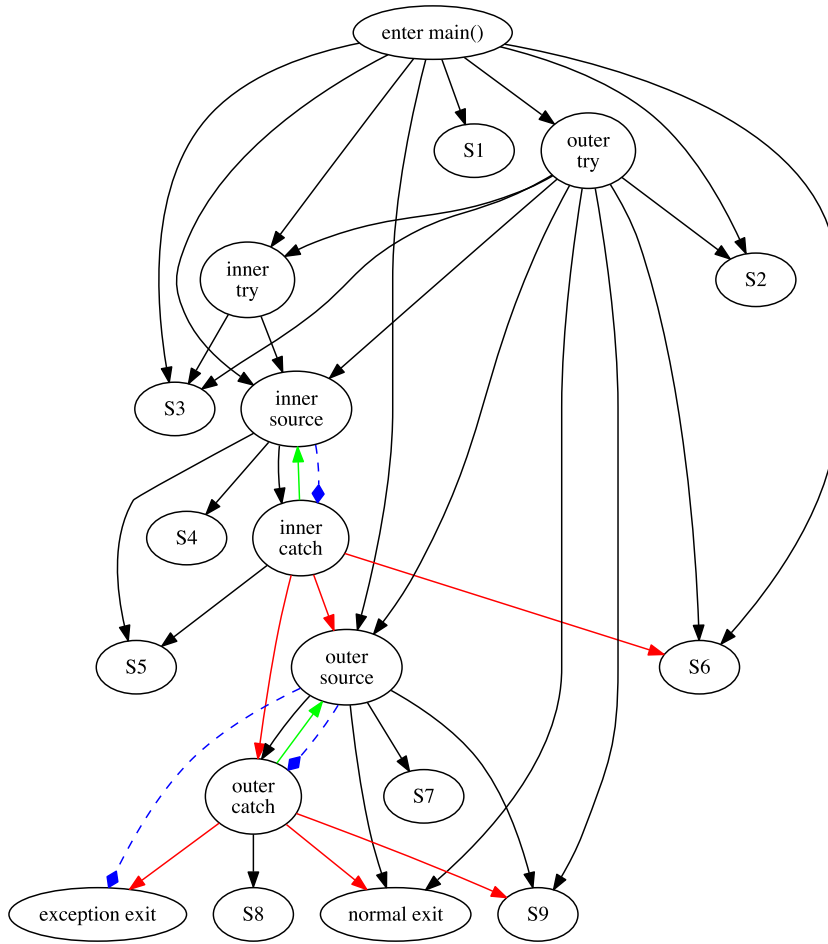
Fig. A.28. The ES-SDG corresponding to the code in Fig. A.24, in the case 4 of Table A.2.

Enter Only the slicing criterion is in the slice.

S1 or `try`(outer) The slice consists of the *Enter* node and the slicing criterion.

S2, `try`(inner), `outer_source` or S6 The slice consists of the *Enter* node, the slicing criterion and the outer `try` node.

S3 or `inner_source` The slice consists of the *Enter* node, the slicing criterion and both `try` nodes (inner and outer).

S4 or `catch`(inner) The slice consists of `inner_source`'s slice, plus the slicing criterion.

S5 The slice consists of inner `catch`'s slice, plus the slicing criterion.

S7 or `catch`(outer) The slice consists of `outer_source`'s slice, plus the slicing criterion.

S8, S9, *normal exit* or *exception exit* The slice consists of outer `catch`'s slice, plus the slicing criterion.

Observe how the data dependency that reaches *exception exit* is the reason for the inclusion of the appropriate exception source. Additionally, notice how the inner `catch` will only be included when (i) the inner exception source and (ii) any instruction after the inner `try-catch` are simultaneously present in the slice; with (ii) being any of S6, `outer_source`, S7, outer `catch`, S8, S9 or any of the *exit* nodes.

A.2.5. Case 5

Consider the case when the exceptions produced in the inner source are completely caught in the outer `catch`, and those produced in the outer source are not completely captured. The corresponding ES-SDG for this case is shown in Fig. A.29. The slices produced by selecting each node as the slicing criterion are the following:

Enter Only the slicing criterion is in the slice.

S1 or `try`(outer) The slice consists of the *Enter* node and the slicing criterion.

S2 or `try`(inner) The slice consists of the *Enter* node, the slicing criterion and the outer `try` node.

S3 or `inner_source` The slice consists of the *Enter* node, the slicing criterion and both `try` nodes (inner and outer).

S4 or `catch`(inner) The slice consists of `inner_source`'s slice, plus the slicing criterion.
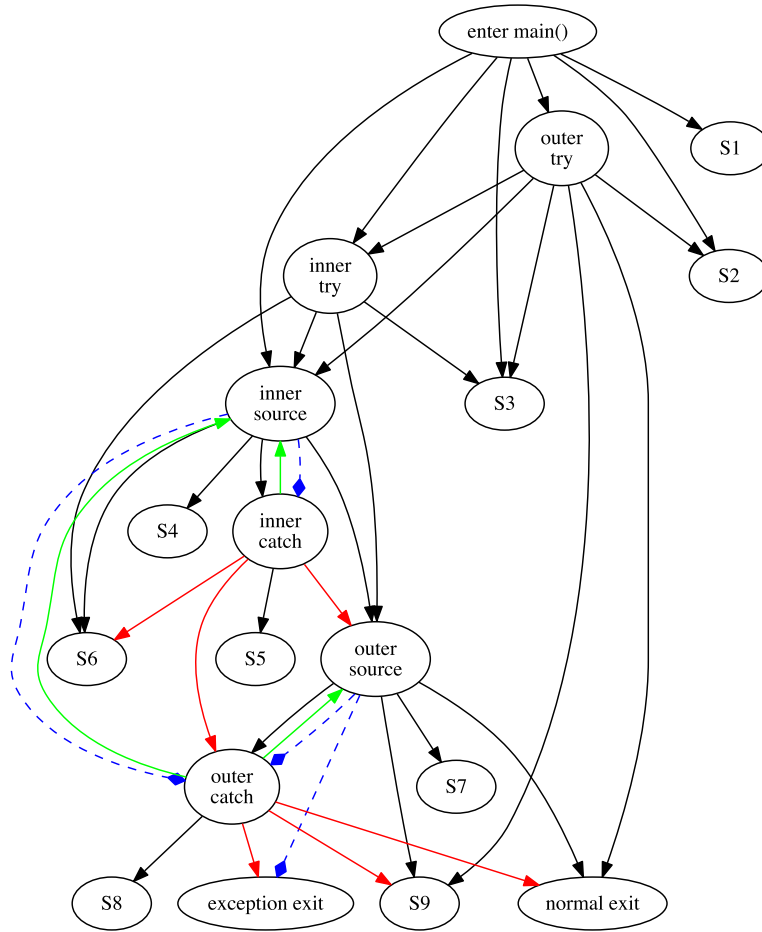
Fig. A.29. The ES-SDG corresponding to the code in Fig. A.24, in the case 5 of Table A.2.

S5, `outer_source` or S6 The slice consists of inner `catch`'s slice, plus the slicing criterion.
S7 or `catch` (outer) The slice consists of `outer_source`'s slice, plus the slicing criterion.
S8, S9, *normal exit* or *exception exit* The slice consists of outer `catch`'s slice, plus the slicing criterion.

Similarly to the previous case (4), the inclusion of *exception exit* is done via data dependencies. Notice how there is no data dependence between the inner source and *exception exit*, because the value thrown there cannot reach the exit, as it is completely captured by the outer `catch`.

A.2.6. Case 6

Consider the case when the exceptions produced in the inner and outer sources are not completely caught in any `catch`. The corresponding ES-SDG for this case is shown in Fig. A.30. The slices produced by selecting each node as the slicing criterion are the following:

Enter Only the slicing criterion is in the slice.
S1 or `try` (outer) The slice consists of the *Enter* node and the slicing criterion.
S2 or `try` (inner) The slice consists of the *Enter* node, the slicing criterion and the outer `try` node.
S3 or `inner_source` The slice consists of the *Enter* node, the slicing criterion and both `try` nodes (inner and outer).
S4 or `catch` (inner) The slice consists of `inner_source`'s slice, plus the slicing criterion.
S5, `outer_source` or S6 The slice consists of inner `catch`'s slice, plus the slicing criterion.
S7 or `catch` (outer) The slice consists of `outer_source`'s slice, plus the slicing criterion.
S8, S9, *normal exit* or *exception exit* The slice consists of outer `catch`'s slice, plus the slicing criterion.

Notice that the only difference between the ES-SDG for Case 6 (Fig. A.30) and Case 5 (Fig. A.29) is the addition of the data dependence between the inner exception source and *exception exit*.
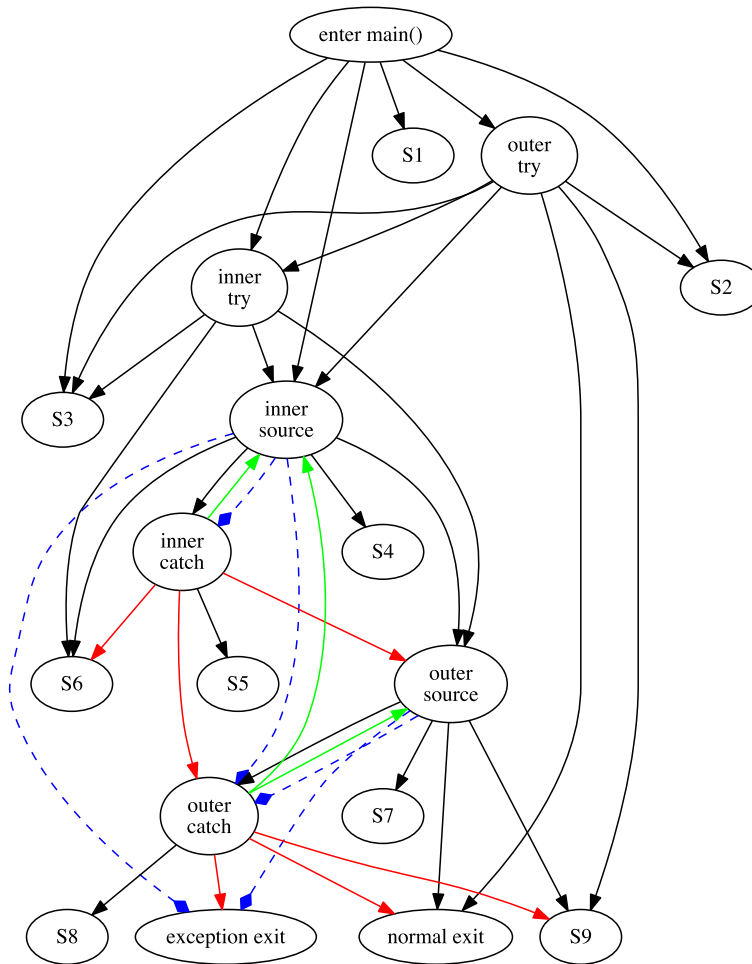
Fig. A.30. The ES-SDG corresponding to the code in Fig. A.24, in the case 6 of Table A.2.

References

[1] Matthew Allen, Susan Horwitz, Slicing Java programs that throw and catch exceptions, SIGPLAN Not. 38 (10) (June 2003) 44–54.

[2] Thomas Ball, Susan Horwitz, Slicing programs with arbitrary control-flow, in: Proceedings of the First International Workshop on Automated and Algorithmic Debugging, AADEBUG '93, Springer-Verlag, London, UK, 1993, pp. 206–222.

[3] Richard A. DeMillo, Hsin Pan, Eugene H. Spafford, Critical slicing for software fault localization, SIGSOFT Softw. Eng. Notes 21 (3) (May 1996) 121–134.

[4] Jeanne Ferrante, Karl J. Ottenstein, Joe D. Warren, The program dependence graph and its use in optimization, ACM Trans. Program. Lang. Syst. 9 (3) (1987) 319–349.

[5] Andy Georges, Dries Buytaert, Lieven Eeckhout, Statistically rigorous Java performance evaluation, SIGPLAN Not. 42 (10) (October 2007) 57–76.

[6] Ákos Hajnal, István Forgács, A demand-driven approach to slicing legacy COBOL systems, J. Softw. Maint. 24 (1) (2012) 67–82.

[7] Susan Horwitz, Thomas Reps, David Binkley, Interprocedural slicing using dependence graphs, ACM Trans. Program. Lang. Syst. 12 (1) (1990) 26–60.

[8] S. Jiang, S. Zhou, Y. Shi, Y. Jiang, Improving the preciseness of dependence analysis using exception analysis, in: 2006 15th International Conference on Computing, IEEE, Nov 2006, pp. 277–282.

[9] H. Jie, J. Shu-juan, H. Jie, An approach of slicing for object-oriented language with exception handling, in: 2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC), Aug 2011, pp. 883–886.

[10] Sumit Kumar, Susan Horwitz, Better slicing of programs with jumps and switches, in: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002), in: Lecture Notes in Computer Science (LNCS), vol. 2306, Springer, 2002, pp. 96–112.

[11] Anirban Majumdar, Stephen J. Drape, Clark D. Thomborson, Slicing obfuscations: design, correctness, and evaluation, in: Proceedings of the 2007 ACM Workshop on Digital Rights Management, DRM '07, ACM, New York, NY, USA, 2007, pp. 70–81.

[12] Prakash Prabhu, Naoto Maeda, Gogul Balakrishnan, Interprocedural exception analysis for C++, in: Proceedings of the 25th European Conference on Object-Oriented Programming, ECOOP'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 583–608.

[13] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, Matthew B. Dwyer, A new foundation for control dependence and slicing for modern program structures, ACM Trans. Program. Lang. Syst. 29 (5) (August 2007) 27–es.

[14] S. Sinha, M.J. Harrold, Analysis of programs with exception-handling constructs, in: Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272), IEEE, Nov 1998, pp. 348–357.

[15] Mark Weiser, Program slicing, in: Proceedings of the 5th International Conference on Software Engineering (ICSE '81), IEEE Press, Piscataway, NJ, USA, 1981, pp. 439–449.