Program Slicing with Exception Handling

Carlos Galindo
cargaji@vrain.upv.es
VRAIN
Universitat Politècnica de València
Valencia, Spain

Sergio Pérez
serperu@dsic.upv.es
VRAIN
Universitat Politècnica de València
Valencia, Spain

Josep Silva
jsilva@dsic.upv.es
VRAIN
Universitat Politècnica de València
Valencia, Spain

Abstract
Program slicing is a technique for program analysis and transformation with many different applications such as program debugging, program specialization, and parallelization. The system dependence graph (SDG) is the most commonly used data structure for program slicing. In this paper, we show that the presence of exception-handling constructs can make the SDG produce incorrect and sometimes even incomplete slices. We showcase the instances of incorrectness and incompleteness and we propose a framework for correctly handling exception-related instructions, which includes representation of all possible exception throwing and catching mechanisms, and a new kind of control dependence: conditional control dependence; which produces more precise slices in the presence of catch statements.

Keywords: program slicing, exception handling, system dependence graph, conditional control dependence

ACM Reference Format:

1 Introduction
Program slicing [14] 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 at a given statement, called a slicing criterion. Program slicing has many practical applications, such as debugging [3], program specialization [10], software maintenance [4], etc. Initially, program slicing was defined for the imperative programming paradigm, but now it can be used with practically all programming paradigms. The most popular data structure used in program slicing is the system dependence graph (SDG), introduced in the late 1980s by Horwitz et al. [5]. It represents statements as nodes and the dependencies between them as arcs, so that the slice can be produced by traversing the graph starting from the slicing criterion. Just as program slicing, the SDG and its underlying elements have been extended to include modern programming languages and their features, such as non-terminating programs [12] or arbitrary control flow [2].

Exception handling is a common feature, present in most modern programming languages. There are several approaches to program slicing with exceptions, but all of them focus on a specific language, such as Java or C++. In reality, the instructions and constructs used in exception handling are quite similar across modern programming languages, with the notable exception of Go*, which purposefully does not include an exception management system and instead relies on early error reporting and panics for important errors.

1.1 Motivation
Precisely due to the similarity between programming languages, the inclusion of exception-handling instructions in program slicing techniques is very similar such that most publications on program slicing with exceptions are generally applicable regardless of the language they are based on. One common approach is the one proposed by Allen and Horwitz [1], which in turn extended Sinha’s proposal [13]. It is arguably the basis used in most publications in the area of exception-aware program slicing. It supports 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 1.1.

Example 1.1 (Incompleteness when slicing try–catch constructs in [1]). Consider the Java program shown in Figure 1a, in which method f is the entrypoint. Two exceptions are thrown, one per call to g, but only the first is captured. Program slicing allows us to identify what parts of the program can lead to the activation of the exception by just selecting line 11 and an empty set of variables as the slicing

*For more information on Go’s design choices regarding exceptions, see https://golang.org/doc/faq#exceptions (retrieved May 2020).
We present our solution as a set of modifications to the \textit{ACFG} which stands for "exception-sensitive", so the \textit{ACFG} becomes the \textit{ES-ACFG}. As it can be seen in Figure 1b, the ES-ACFG would only remove line 5 from the code (see Figure 1c). As it can be seen, Allen and Horwitz do not include the catch statement, despite being necessary to execute the second call to \texttt{g}. Thus, the slice produced by this approach is not complete.

The source of this error is that in Allen and Horwitz’s approach \textit{catch} blocks are included only in a specific case: the slicing criterion is or requires a variable defined inside the \textit{catch} block. This only happens when a statement of the \textit{catch} block is included in the slice and, consequently, control dependencies force the \textit{catch} itself to be included too. Unfortunately, this is insufficient, since it does not capture the complex control dependency involved in using \textit{catch} blocks. This counter example shows that even empty \textit{catch} blocks may be necessary in the slice.

1.2 Contributions

The main contribution of this work is a new approach to program slicing with exception handling which is proven complete in all cases (e.g., it solves the previous motivating problem). Moreover, the slices produced are strictly more precise than previous approaches. It is applicable to most modern programming languages, such as Java, C++, and JavaScript, among others. Our approach extends the techniques proposed by Allen and Horwitz [1], while also using other improvements regarding control dependence introduced by Kumar and Horwitz [9].

The rest of this paper is structured as follows: Section 2 describes our proposal, Section 3 compares our solution to similar approaches and other proposals in the recent past, and Section 4 summarizes our results.

2 Slicing exceptions

We present our solution as a set of modifications to the standard construction of the SDG. Our baseline employs basic improvements to control dependence computation such as the augmented control-flow graph (ACFG) [2] and the pseudo-predicate program dependence graph (PPDG) [9]. We organize our modifications in the different phases of creation of a slice: code to ACFG to PPDG to SDG, and finally traversal of the SDG. In order to clearly differentiate between each version of the graph, our graphs are prefixed by ‘ES-’, which stands for “exception-sensitive”, so the ACFG becomes the \textit{ES-ACFG}, the PPDG becomes the \textit{ES-PPDG}, and the SDG becomes the \textit{ES-SDG}.

2.1 Modifications to the ACFG to create the \textit{ES-ACFG}

In this section we describe compositionally how to construct any \textit{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 \textit{ES-ACFG} has three kinds of nodes: statements, which have only one outgoing arc; predicates, which have multiple outgoing arcs, though it is typically two labeled \textit{true} and \textit{false}; and pseudo-predicates, which have two outgoing arcs labeled \textit{true} and \textit{false}. The \textit{false} arc in a pseudo-predicate represents a non-executable step.

Most instructions of the ACFG keep their traditional representation, but there are six constructs that need to be modified to properly account for exception handling: procedure declarations, procedure calls and all structures that cause or catch exceptions. The rest of this subsection explains in detail these instructions and their correct representation. Figure 2 showcases a simple generalized version of each instruction. Arcs are not labeled for simplicity, but non-executable arcs are displayed with a dashed arc. We often use \textit{true} and \textit{false} to refer to executable and non-executable arcs, respectively.

Unconditional exception sources are instructions whose execution will always result on an exception being thrown or activated. They are represented as a pseudo-predicate, as a return statement would be. The \textit{true} arc will be connected to the first catch instruction that can capture it, or otherwise to the exception \texttt{exit}. The \textit{false} arc will be connected to the instruction that would be executed if the pseudo-predicate failed to throw the exception. Figure 2a shows a scheme.

Conditional exception sources are instructions whose execution may activate an exception. They have the same representation as unconditional sources, but instead of being pseudo-predicates, they are predicates; to account for the fact that the exception may or may not be thrown. Figure 2b shows an example, displaying the change from pseudo-predicate to predicate.

Exception catching structures

\texttt{try} is represented as a pseudo-predicate, with its \textit{true} arc connected to the first instruction within its body, and its \textit{false} arc connected to the first instruction after the whole structure. A scheme is shown in Figure 2c.

\texttt{catch} is represented either as a pseudo-predicate or a predicate, depending on whether all exception sources that are connected to it will be caught or not, respectively. In both cases, its \textit{true} arc is connected to the first instruction in its body, and its \textit{false} arc is connected to the next catch node that will catch one of the exceptions or otherwise to the exception \texttt{exit} node. Both cases can be seen in Figures 2d and 2e.
public void f() {
    try {
        g();
        e.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    }
    g();
}

public void g() {
    throw new Exception();
}

(a) The program
(b) Allen and Horwitz’s slice
(c) The correct slice

Figure 1. Java program that throws two exceptions but captures only the first one, its slices, and one SDG representation.

Procedures that may throw or propagate exceptions
Calls to these procedures are represented as: one procedure call node, where the arguments are evaluated and the call made; one normal return node, which is reached only if no exception was propagated through the procedure; and one exception return node, which is reached when an uncaught exception was thrown in the method call.

The procedure call is 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.

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

(d) Allen and Horwitz’ SDG associated with Figure 1b.

Figure 2. ACFGs of the structures relevant to exception handling.

(a) Unconditional exception source
(b) Conditional exception
source
(c) try instruction
(d) catch predicate
(e) catch pseudo-predicate
(f) Procedure call with exceptions
(g) Procedure declaration exit with exceptions
The two return nodes contain assignments for modified global variables and parameters passed by reference. A scheme is shown in Figure 2f.

Procedure declarations with exceptions The Exit node is split into three nodes: normal exit, exception exit and exit.

normal exit This performs the function of the old Exit node. It is represented as a statement, whose arc is connected to Exit.

exception exit This is the equivalent to normal exit, but for exceptions.

Exit A sink node, guaranteeing the common requirement of CFGs having only one sink.

When there are formal-out nodes, they are moved to the specialized exit nodes, for increased precision.

Figure 2g shows exit of a procedure with exceptions.

An additional variable must be tracked throughout procedures that throw exceptions: the active exception; which is declared in exception sources and used in exception exit and catch nodes.

2.2 Modifications to the PPDG to create the ES-PPDG

Example 1.1 reveals that the SDG proposed by Allen and Horwitz can generate incomplete slices: catch blocks are not correctly represented. A catch block is a statement that is only relevant if the program execution does not occur normally. For this reason, the control dependencies they induce are slightly different from the ones generated by other statements. Instead of influencing other statements with their presence, it is their absence what may lead to a non-desired behaviour. We can illustrate this with the code in Figure 3 and considering three different slicing scenarios that allow us to analyse how does the presence or absence of the catch statement affect the 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. The slice would be lines 1, 3, and 6.

2. Only \( \log(x) \) is part of the slice. If only \( \log(x) \) is in the slice, although the catch statement controls it, since there is no possible statement inside the try-catch block to raise an exception that the catch captures, the catch statement does not influence the execution of \( \log(x) \). The slice would be lines 1, 5, and 6.

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.

These scenarios reveal a new kind of control dependence which is conditional. 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. 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 2.1 (Conditional control dependence). Let \( P = (N, A) \) be a PPDG. We say a node \( a \in N \) is conditional control dependent on a pair of nodes \( b, c \in N \), if the presence of \( a \) allows the execution of \( c \) when \( b \) is executed and the absence of \( a \) prevents it.

Algorithm 1 describes the process to transform a PPDG into an ES-PPDG through the definition of the conditional control dependency sets CC1 and CC2. This algorithms makes use of 4 different methods with descriptive names. For instance, function getBlockInstructs/1, that receives a catch node as argument, returns a set with all the instructions into the catch block. Additionally, set CatchNodes contains all the catch nodes of the graph. In Algorithm 1 the use of \( A' \) represents the reflexive and transitive closure of \( A \).

Algorithm 1 analyses every catch node independently and divides its processing in two steps, the generation of the CC1 arcs, and the generation of the CC2 arcs. In the first part, it selects every outgoing control arc from a catch node and move it from the \( A \) set to the \( A_{cc1} \) set if this statement it points to is not inside the catch block. In the second part, all the nodes inside the try block are selected one by one, and two conditions decide whether a CC2 arc needs to be added to the graph or not: (i) the node represents a conditional or unconditional exception source and (ii) there is a control path in the PPDG from the exception source to the catch node which is pseudo-predicate free. If these two conditions are fulfilled, an arc from the catch node to the exception source is added to the set \( A_{cc2} \). Finally, all the sets of arcs are put together in the set \( A' \) and the final ES-PPDG is returned.

2.3 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 dependencies are generated. The creation of call, parameter-in, parameter-out and summary arcs is the same as in the SDG.
The main difference between the common SDG and the ES-SDG is the treatment of the different possible exit contexts. Every ES-PPDG may have two different Exit nodes: normal exit and exception exit. For this reason, the ES-SDG features an additional kind of arc: the return arc, which connects a exit node in the declaration to its corresponding return node in the call. These can be seen in Figure 4, where dotted arcs connect each exit to their corresponding return counterparts.

### 2.4 Slicing the ES-SDG

The ES-SDG introduces various structural changes and a new kind of arcs: the conditional control dependence arcs. Therefore, we need to determine how this new arcs are treated by the slicing algorithm. The new graph traversal is based on the slicing algorithm proposed by Horwitz et al. in [6] but with some new considerations due to all the new introduced elements: return arcs, conditional arcs, and new instructions being handled as pseudo-predicates. It can be summarized with the next 4 rules:

1. The graph is traversed in two sequential passes. In the first pass, output (param-out and return) arcs are ignored; and in the second pass, input (param-in and call) arcs are ignored. Each pass ends when there are no more new arcs to traverse.
2. If a node is reached by another conditional type of type , such that , then . If included in the slice, no arcs will be traversed from , unless it is reached via another non-conditional arc.
3. Conditional arcs of type CC1 are transitive, even when the intermediate node is not included in the slice. As an example, if , then . , even when is not in the slice.
4. Control dependency arcs that reach a node will not be traversed if is a pseudo-predicate and has only been reached via control dependency arcs. Conditional control dependency arcs are not considered control dependency arcs for this matter. This is based on the traversal restriction introduced by Kumar and Horwitz [9] for the slicing of the PPDG.

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 linear time.

Example 2.2 shows the ES-SDG for the motivating example, sliced with the same criterion.

**Example 2.2** (A correctly generated slice for the program in Example 1.1). If we apply our algorithm to the problem shown in our motivating example (Example 1.1), we obtain the ES-SDG shown in Figure 4. If we then choose as slicing criterion (throw, ) in line 11, the Enter g() node is included, 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, so the exceptions generated by g’ s first call can be caught and g’ s second call can be executed.
3 Related work
As we have already mentioned, our work is based on the ACFG [2] and the PPDG [9]. 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 dependencies they needed. Despite this, they failed to account for the conditional need of catch statements, even when in the original program no exception will escape from it, and therefore, from a pure control flow approach, the whole try-catch block cannot influence any instruction after it.

Later, Jiang et al. [7] 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 dependencies from procedure calls to catch nodes, but did not generalize that concept to all exception sources and usages. Other approaches include Prabhu et al. [11], which centered around the exception system of C++, and its specific quirks and design choices; and Jie et al. [8], 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.

4 Conclusions
Program slicing is a powerful software analysis technique, powered by the system dependence graph, a directed graph that represents statements and their dependencies. In this paper, we have presented a new approach for program slicing with exception handling, based on previous publications and focusing on 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 correctness of slices by using a new notion of control dependency called conditional control dependency, which allows for the conditional inclusion of catch statements: they are only added to the slice when it contains both a statement that requires an exception e to be caught, and a statement that throws e. Thus, we limit the inclusion of try-catch instructions and exception sources to the minimum necessary to generate complete slices.

References