UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

DOCTORAL THESIS

Program Slicing for Modern Programming Languages

Author
Carlos Santiago
GALINDO JIMÉNEZ

Supervisor
Dr. Josep
SILVA GALIANA

Valencian Research Institute for Artificial Intelligence (VRAIN)
Departamento de Sistemas y Computación (DSIC)

September, 2024

Para mis padres, por todo el amor que me han dado.

"Take wrong turns. Talk to strangers. Open unmarked doors. And if you see a group of people in a field, go find out what they are doing. Do things without always knowing how they'll turn out."

Randall Munroe

Abstract

Producing efficient and effective software is a task that has remained difficult since the advent of computers. With every improvement on hardware and developer tooling (e.g., compilers and checkers), the demand for software has increased even further. This means that auxiliary analyses have become integral in developing complex software systems.

Program slicing is a static analysis technique that gives answers to *What parts of the program can affect a given statement?*, and similar questions. Its main application is debugging, as it can reduce the amount of code on which a programmer must look for a mistake or bug. Other applications include program parallelization and specialisation, program comprehension, and software maintenance. Lately, it has mostly been applied as a pre-processing step in other expensive static analyses, to lower the size of the program and thus the analyses' runtime. The most popular data structure in program slicing is the system dependence graph (SDG), which represents statements as nodes and dependences as arcs between them. The two main types of dependences are control and data dependences, which encapsulate the control and data flow throughout every possible execution of a program.

Programming languages are an ever-expanding subject, with new features coming to new releases of popular and up-and-coming languages like Python, Java, Erlang, Rust, and Go. However, program slicing was originally defined for (and has been mostly focused on) imperative programming languages. Even then, some popular elements of the imperative paradigm, such as arrays and exceptions do not have an efficient or sometimes complete representation in the SDG. Other paradigms, such as functional or object-oriented also suffer from partial support in the SDG.

This thesis presents improvements for common programming constructs, and its contributions are split into control and data dependence. For the former, we (i) specify a new representation of *catch* statements, along with a full description of other exception-handling constructs. We also (ii) analyse the current state-of-the-art technique for unconditional jumps (e.g., *break* or *return*), and show the risks of combining it with other popular techniques. Then, we focus on concurrency, with a (iii) formalisation of a reversible, causal-consistent debugger for CSP specifications. Switching to data dependences, we focus our contributions on making existing techniques context-sensitive (i.e., more accurate in the presence of routines or functions). We explore the data dependences involved in shared-memory concurrent programs, (iv) redefining interference dependence to make it context-sensitive. Afterwards, we take a small detour to (v) explore the decidability of various data analyses

on programs with (and without) complex data structures and routine calls. Finally, we (vi) extend our previous work on slicing complex data structures to combine it with tabular slicing, which provides context-sensitivity.

Additionally, throughout this thesis, multiple supporting software libraries have been written or extended with the aforementioned improvements to program slicing. These have been used to provide empirical evaluations, and are available under libre software licenses, such that other researchers and software developers may extend or contrast them against their own proposals. The resulting tools are two program slicers for Java and Erlang, and a causal-consistent reversible debugger for CSP.

Resumen

Producir *software* eficiente y efectivo es una tarea que parece ser tan difícil ahora como lo era para los primeros ordenadores. Con cada mejora de *hardware* y herramientas de desarrollo (como son compiladores y analizadores), la demanda de producir software más rápido y más complejo ha ido aumentando. Por tanto, todos estos análisis auxiliares ahora son una parte integral del desarrollo de programas complejos.

La fragmentación de programas es una técnica de análisis estático, que da respuesta a «¿Qué partes del programa pueden afectar a esta instrucción?», y otras preguntas similares. Su aplicación principal es la depuración de programas, porque puede acotar la zona de código a la que el programador debe prestar atención mientras busca la causa de un error. También tiene otras muchas aplicaciones, como pueden ser la paralelización y especialización de programas, la comprensión de programas y el mantenimiento. En los últimos años, su uso más común ha sido como preproceso a otros análisis con alto coste computacional, para reducir el tamaño del programa a procesar, y, por tanto, el tiempo de ejecución de estos. La estructura de datos más popular para fragmentar programas es el *system dependence graph* (SDG), un grafo dirigido que representa las instrucciones de un programa como vértices, y sus dependencias como arcos. Los dos tipos principales de dependencias son las de control y las de datos, que encapsulan el flujo de control y datos en todas las ejecuciones posibles de un programa.

El área de lenguajes de programación está en eterno cambio, ya sea por la aparición de nuevos lenguajes o por el lanzamiento de nuevas características en lenguajes existentes, como pueden ser Python, Java, Erlang, Rust o Go. Sin embargo, la fragmentación de programas se definió originalmente para (y está aún enfocada a) el paradigma imperativo. Aun así, hay características populares en lenguajes imperativos, como las *arrays* y las excepciones, que aún no tienen una representación eficiente y/o completa en el SDG. Otros paradigmas, como el funcional o el orientado a objetos, sufren también de un soporte parcial en el SDG.

Esta tesis presenta mejoras para construcciones comunes en la programación moderna, dividiendo contribuciones en las enfocadas a dependencias de control y las enfocadas a datos. Para las primeras, (i) especificamos una nueva representación de instrucciones *catch*, junto a una descripción completa del resto de instrucciones relacionadas con excepciones. También (ii) analizamos las técnicas punteras para saltos incondicionales (p.e., *break* o *return*), y mostramos los riesgos de combinarlas con otras técnicas para objetos, llamadas o excepciones. A continuación, ponemos nuestra mirada en la concurrencia, con

una (iii) formalización de un depurador de especificaciones CSP reversible y causal-consistente. En cuanto a las dependencias de datos, se enfocan en técnicas sensibles al contexto (es decir, más precisas en presencia de rutinas y sus llamadas). Exploramos las dependencias de datos generadas en programas concurrentes por memoria compartida, (iv) redefiniendo las dependencias de interferencia para hacerlas sensibles al contexto. A continuación, damos un pequeño rodeo por el campo de la indecidibilidad, en el que (v) demostramos que ciertos tipos de análisis de datos sobre programas con (y sin) estructuras de datos complejas son indecidibles. Finalmente, (vi) ampliamos un trabajo previo sobre la fragmentación de estructuras de datos complejas, combinándolo con la fragmentación tabular, que la hace sensible al contexto.

Además, a lo largo de toda la tesis, se han desarrollado o extendido múltiples librerías de código con las mejoras mencionadas anteriormente. Estas librerías nos han permitido realizar evaluaciones empíricas para algunos de los capítulos, y también han sido publicadas bajo licencias libres, que permiten a otros desarrolladores e investigadores extenderlas y contrastarlas con sus propuestas, respectivamente. Las herramientas resultantes son dos fragmentadores de código para Java y Erlang, y un depurador de CSP reversible y causal-consistente.

Resum

La producció de programari eficient i eficaç és una tasca que resulta tan difícil hui dia com ho va ser durant l'adveniment dels ordinadors. Per cada millora de maquinari i ferramentes per al desenvolupament (com poden ser els compiladors i analitzadors), augmenta sovint la demanda de programes, així com la seua complexitat. Com a conseqüència, totes aquestes anàlisis auxiliars esdevenen una part integral del desenvolupament de programari.

La fragmentació de programes és una tècnica d'anàlisi estàtica, que respon a **?**Quines parts d'aquest programa poden afectar a aquesta instrucció?**?**, i altres preguntes similars. L'aplicació principal d'aquesta tècnica és la depuració de programes, per la seua capacitat de reduir la llargària d'un programa sense canviar el seu funcionament respecte a una instrucció que està fallant, delimitant així l'àrea del codi en què el programador busca l'origen de l'errada o *bug*. Tot i això, té moltes altres aplicacions, com la paral·lelització i especialització de programes o la comprensió de programes i el seu manteniment. Durant els darrers anys, l'ús més freqüent de la fragmentació de programes ha sigut com a «preprocés» abans d'altres anàlisis amb un alt cost computacional, per tal de reduir-ne el temps requerit per realitzar-les. L'estructura de dades més popular per fragmentar programes és el *system dependence graph* (SDG), un graf dirigit representant-ne les instruccions d'un programa amb vèrtexs i les seues dependències amb arcs. Els dos tipus principals de dependència són el de control i el de dades, aquests encapsulen el flux de control i dades a totes les possibles execucions d'un programa.

L'àrea dels llenguatges de programació s'hi troba en constant evolució, o bé per l'aparició de nous llenguatges, o bé per noves característiques per als preexistents, com poden ser Python, Java, Erlang, Rust o Go. No obstant això, la fragmentació de programes s'hi va definir originalment (i encara hi està enfocada) per al paradigma imperatiu. Tot i que, també hi trobem característiques populars als llenguatges imperatius, com els arrays i les excepcions, que encara no en tenen una representació eficient i/o completa al SDG. Altres paradigmes, com el funcional o l'orientat a objectes, pateixen també d'un suport reduit al SDG.

Aquesta tesi presenta millores per a construccions comunes de la programació moderna, dividint les contribucions entre aquelles enfocades a les dependències de control i aquelles enfocades a dades. Per a les primeres, (i) hi especifiquem una nova representació d'instruccions *catch*, junt amb una descripció completa de la resta d'instruccions relacionades amb excepcions. També (ii) hi analitzem les tècniques capdavanteres de fragmentació de salts incondicionals (per exemple, *break* o *return*), i hi mostrem els riscs de combinar-ne-les amb altres tècniques per a objectes, instruccions de crida i excepcions.

A continuació, hi posem la nostra atenció en la concurrència, amb (iii) una formalització d'un depurador d'especificacions CSP reversible i causal-consistent. Respecte a les dependències de dades, dirigim els nostres esforços a produir tècniques sensibles al context (és a dir, que es mantinguen precises en presència de procediments o funcions). Hi explorem les dependències de dades generades en programes concurrents amb memòria compartida, (vi) redefinint-ne les dependències d'interferència per a fer-ne-les sensibles al context. Seguidament, (v) hi demostrem la indecidibilitat d'alguns tipus d'anàlisis de dades per a programes amb (i sense) estructures de dades complexes. Finalment, (vi) hi ampliem un treball previ sobre la fragmentació d'estructures de dades complexes, combinant-lo amb la fragmentació tabular, fent-hi-la sensible al context.

A més a més, durant la realització d'aquesta tesi, s'han desenvolupat o estés diverses llibreries de codi amb les millores esmentades prèviament. Aquestes llibreries ens han permés avaluar empíricament alguns dels capítols i també han sigut publicades sota llicències lliures, fet que permet a altres desenvolupadors i investigadors poder estendre-les i contrastar-les, respectivament. Les ferramentes resultants són dos fragmentadors de codi per a Java i Erlang, i un depurador CSP el qual és reversible i causal-consistent.

Agradecimientos

Hace seis años, durante un intercambio académico en la *ETHz*, tuve la oportunidad de asistir a una asignatura que fijaría mi atención en el campo de análisis de *software*: *Compiler Design*, impartida por el profesor T. Gross. Gracias a él, tanto durante la asignatura, como al realizar el trabajo fin de grado, acabé de entender los programas y su funcionamiento, desde el lenguaje de alto nivel en el que se escribe hasta las instrucciones de ensamblador que el ordenador puede entender. Es por esto que, al volver de Suiza y elegir máster, me especialicé en *ingeniería de software*. En la asignatura *Transformación, Validación y Depuración de programas* conocí el concepto de fragmentación de programas (o *program slicing*), y también a Josep Silva, que me ofreció la oportunidad de realizar el trabajo final de máster con él. La última pieza del puzle la aportó Germán Vidal, que ofertaba un contrato de investigación predoctoral.

Muchas son las personas a las que tengo que dar gracias, empezando por el mismo Germán, gracias al cual pude empezar el doctorado con la seguridad de un contrato, integrarme en el grupo MiST de la UPV, explorar si quería hacer tesis doctoral, y elegir tema.

Sin embargo, la clave de que esta tesis haya llegado a buen puerto es Josep Silva. Josep, te tengo que agradecer tu presencia y atención constante (día y noche) durante estos cuatro años, el apoyo incondicional que me has prestado y tu guía y consejos sobre investigación, docencia, administración, publicaciones... Gracias por tus mensajes de madrugada, respondiendo dudas más o menos tontas, por enseñarme a diseñar experimentos y a formalizar pruebas, por emprender proyectos para que pueda continuar mi carrera universitaria después de la tesis. Gracias por tus explicaciones, por tu ética de trabajo excelente y por todas las oportunidades que me has dado para desarrollarme personal y profesionalmente. Contigo tengo una deuda impagable. Gracias, gracias, gracias.

Si bien la dirección de una tesis es fundamental, no quiero olvidarme del resto de profesores e investigadores que han sido un referente y ayuda al navegar estos cuatro años. Gracias a los profesores de los grupos ELP y MiST, que me han acogido y aconsejado, y me han acompañado en diversas conferencias. En concreto, quiero dar las gracias a Marisa, por sus consejos docentes, a Santiago, por ayudarme a navegar las distintas burocracias universitarias, a Alicia y Raúl, siempre atentos a las necesidades de los becarios, y a Salva, siempre dispuesto a compartir su conocimiento sobre la historia de la informática. Gracias a Coen y Pedro Manuel por su experta revisión de esta tesis y sus comentarios.

También son muchos los compañeros que han asomado la cabeza por la puerta del laboratorio, como alumnos, doctorandos o ya doctores. Desde los

almuerzos y cafés más triviales hasta discusiones con una pizarra llena de grafos. Gracias, Adrián, Julia, Víctor, Félix y Julián. Sé que tendréis éxito allá donde vayáis.

No me puedo olvidar tampoco de aquellos que me han acogido en el extranjero. Tras mi aventura de tres meses en el *University College London*, debo agradecer profundamente al profesor Jens Krinke que me abriera las puertas del grupo CREST, y me transmitiera su experiencia investigadora, su maestría de *slicing* y tantas comidas discutiendo cultura y eventos locales. Gracias también a Nicolas, otro experto en *slicing*, Justyna y Federica, que al mismo llegar me incluyeron en el *workshop* que estaban desarrollando, y muchas gracias a Bill, Maria y Mar, que le dieron vida a mi estancia en Londres.

En una tesis hay tiempo para hacer muchas cosas, pero la más común para mí ha sido estar en un laboratorio, rodeado de compañeros excelentes. Ya he mencionado a algunos, pero con estos tengo una deuda mayor. Gracias, Carlos, por la ilusión que traes a problemas que nos vienen un poco grandes a todos. Gracias, Chris, por tu mente siempre llena de café e ideas, capaz de resolver cualquier cuestión que se plantea; un verdadero patito de goma. Gracias, Damián, por tantas conversaciones que empiezan con comandos obscuros en LaTeX y no tienen fin. Y finalmente, gracias, Sergio. Gracias por ir un paso por delante, abriendo camino. Aunque signifique trabajar ocho años en un artículo para que se publique justo después. Pocas personas pueden disfrutar de un compañero así. No solo por la cercanía de nuestras tesis doctorales, y las simbiosis entre ellas, sino por la entrega y pasión por todo: tanto el trabajo como el descanso, los viajes y los almuerzos, los experimentos y las revisiones. En ti he visto lo que significa ser investigador, qué contiene una buena tutoría, cómo alentar a los alumnos a aprender. En definitiva, te echaré de menos... hasta los próximos *scaquing games*.

He de agradecer también a mis amigos, que han sufrido pacientemente cuatro años de un Carlos intermitente, con explicaciones ininteligibles sobre "tijeras automáticas para programas", desapariciones que coincidían misteriosamente con entregas de artículos, y frustraciones diversas. En especial quiero dar las gracias al resto de catequistas de San Leandro, y también al equipo de Cáritas, que siempre me han apoyado.

Para terminar, quiero dar las gracias a mis padres, que me han empujado a dar siempre lo mejor de mí, a esforzarme todo lo posible y a elegir el camino más adecuado. Es gracias a ellos que he llegado tan lejos. Desde las primeras letras aprendiendo a leer, pasando por las redacciones que no pasaban de media página. Gracias por darme vuestro amor y educarme. Os quiero, aunque no siempre lo parezca. Gracias a mis hermanos, Lola, Jose, Inés y Miguel, porque con vosotros he aprendido a trabajar en equipo, ya sea cocinando, jugando o limpiando. Gracias a mis abuelos, Pepe y Ana, Pascual y Loli, que

han sido ejemplo de perseverancia y amor incondicional. Y, especialmente, me gustaría agradecer a Rosalía, cuyo amor me ha acompañado ya durante doce años. Gracias a ti, este libro existe.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ACFG	**A**ugmented **C**ontrol-**f**low **G**raph (Definition 4.1)
APDG	**A**ugmented **P**rogram **D**ependence **G**raph (Definition 4.2)
CE-PDG	**C**onstrained-**e**dges **P**rogram **D**ependence **G**raph (Section 8.1)
CFG	**C**ontrol-**f**low **G**raph (Definition 2.5)
CSP	**C**ommunicating **S**equential **P**rocesses (Section 5.1)
DDA	**D**ata **D**ependence **A**nalysis (Definition 7.2)
DDG	**D**ata **D**ependence **G**raph (Section 2.3)
DFA	**D**ata-**f**low **A**nalysis (Definition 7.1)
ES-CFG	**E**xception-**s**ensitive **C**ontrol-**f**low **G**raph (Section 3.2.1)
ES-PDG	**E**xception-**s**ensitive **P**rogram **D**ependence **G**raph (Section 3.2.2)
ES-SDG	**E**xception-**s**ensitive **S**ystem **D**ependence **G**raph (Section 3.2.3)
PDG	**P**rogram **D**ependence **G**raph (Definition 2.10)
P-PCP	**P**arentesis-**P**ost **C**orrespondence **P**roblem (Definition 7.4)
PPDG	**P**seudo-predicate **P**rogram **D**ependence **G**raph (Definition 4.4)
RDA	**R**eps' **D**ata **A**nalysis (Definition 7.3)
SDG	**S**ystem **D**ependence **G**raph (Definition 2.11)
tCFG	**T**hreaded **C**ontrol-**f**low **Graph** (Section 6.1)
TNG	**T**opological **N**umber **G**raph (Section 6.1)
tPDG	**T**hreaded **P**rogram **D**ependence **G**raph (Section 6.1)
tSDG	**T**hreaded **S**ystem **D**ependence **G**raph (Section 6.3)

Part I

What is Program Slicing?

Chapter 1

Introduction

1.1 Motivation

The seed for this doctoral thesis started almost five years ago, with a promise:

Program slicing can help you debug programs by excluding or otherwise obscuring code that is not relevant to your breakpoints.

The promise's author shall remain anonymous, but it rang specially true for me, a then-Master's student. Even though modern debuggers have many powerful and useful features, they still cannot achieve what program slicing is designed to do: separate the wheat from the chaff, remove any statement that is irrelevant to the task at hand.

This should not be necessary for expert programmers. After all, that is why procedures, functions, objects, and other forms of compartmentalization are for. Each section of a program should focus on performing a specific task well, and the composition of simpler structures can result in systems that can do complex tasks perfectly. Sadly, we do not live in this idealized world. Time and budget constraints force programmers to take shortcuts, build the simplest structure that can solve the given problem, without worrying about compartmentalization and maintainability. This is often called technical debt, because of the long-tail effects it can have on a software project. Left unmanaged, technical debt can accumulate and make the development of new features or fixing bugs prohibitely expensive. As Ward Cunningham put it back on 1992 [17]:

> Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. [...] The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise.

Program slicing promises to alleviate this ever-increasing complexity problem, and many others. Because, at its core, program slicing is a simplification tool. The user selects a point of interest, and all code that is irrelevant to it is removed, leaving only relevant code, the so-called slice.

- *Is debugging your program too complicated due to procedures that try to do too much?* Slice the program at the location of the error to simplify it. Or, better yet, use program slicing to separate intertwined but independent tasks to distinct procedures.
- *Is your static analysis too slow to run? Perhaps due to an exponential cost?* Produce a slice and lower the time required for analysis.
- *Does your software take advantage of multicore architectures? No?* Try slicing multiple times to parallelise a previously-sequential process.
- *Is your version control repository filled with complex changesets or commits?* Use a program slicer to automatically detect smaller, independent changes and split them accordingly.

Program slicing has a wide range of applications, as documented on Silva's survey [99], but remains an unknown technique to most of its potential users.

Alas, there is no modern program slicer available. The commercial analysis suite *Codesonar*, developed by *Grammatech*, used to include a slicer. *Indus-Kaveri*, an open source Java slicer with support for concurrency, was distributed 10 years ago as a plugin for the Eclipse IDE. However, support ended for that release of Eclipse and the university forge that hosted the source code shut down[1]. The Watson Library for Analysis (WALA), an IBM project, includes slicers for Java bytecode and JavaScript. There seem to be more defunct slicers than active ones. To be best of our knowledge, no IDE offers slicing as a core feature or a plugin.

The problem is theoretical. Originally, Weiser [106] defined program slicing around a simple, imperative paradigm programming language. It shines when handling branching and looping control structures, and classic variable assignment and re-assignment. However, current programming languages are much more complex and feature much more paradigms than that. Many works have built up from Weiser's seminal work to adapt program slicing to existing and upcoming language features. Still, there are some areas in which slicing does not produce correct or precise enough results, which makes it useless for most applications.

This thesis tries to produce new theory to advance the field, and useful program slicers that can fulfil that promise. It is not revolutionary, a panacea, but a small step towards better program slicers, available and useful to all.

[1] A partial archive of the source code can be found at `https://github.com/rvprasad/Indus_archive`.

```
1  read(n);
2  sum = 0;
3  prod = 1;
4  i = 0;
5  while (i < n) {
6    sum  += i;
7    prod *= i;
8    i++;
9  }
10 write(sum);
11 write(prod);
```

(A) The original program.

```
1  read(n);
2  sum = 0;
3  prod = 1;
4  i = 0;
5  while (i < n) {
6    sum  += i;
7    prod *= i;
8    i++;
9  }
10 write(sum);
11 write(prod);
```

(B) The program slice.

FIGURE 1.1: A classic program slicing example program [104], sliced w.r.t. $\langle 10, sum \rangle$. Grey code has been removed (sliced).

1.2 Program Slicing

The previous section describes program slicing in broad strokes, but the question remains unanswered: *What is program slicing?* Program slicing [99, 104] is a program analysis technique to answer the question: "Which statements affect the value of a variable v at a program point p?" The *slice* (or *program slice*) is the answer to this question, expressed as a set of nodes or a trimmed version of the original program. The point of interest in the program, $\langle p, v \rangle$, is the so-called *slicing criterion*.

Example 1.1 (Slicing a simple program)**.**
Consider the code shown in Fig. 1.1a, which computes a sequence of sums and multiplications based on a user-provided input, `n`. Both operations are independent, so program slicing can separate them. If we select $\langle 10, sum \rangle$ as our slicing criterion (line 10, variable *sum*, underlined), we obtain the slice shown in Fig. 1.1b. The slice, in black, computes the same value for *sum* as the original program, and excludes three statements that are irrelevant to the selected criterion (those that concern *prod*).

Slices can be computed in a variety of ways. Weiser's first proposal [106] was the use of data-flow equations to determine which statements must be included in the slice and which can be safely removed. Binkley et al. [10] proposed ORBS (Observation-Based Slicing), a language-independent technique which randomly removes lines of a program and uses testing to determine whether the new program is a valid slice or not. The most commonly used technique, however, relies on dependence graphs, which model statements as nodes and program dependences as directed arcs, and then can produce slices

as a reachability problem over the graph. This method, by Ottenstein and Ottenstein [81], started with the program dependence graph (PDG), which contains the nodes and dependences of a single procedure in a program. Later, Horwitz et al. [53] generalized the method for complete systems (composed of collections of procedures) by introducing the system dependence graph (SDG), whose main improvement was the ability to differentiate between multiple calls to the same procedure (i.e., it was context-sensitive).

1.3 Contributions and goals

This thesis' main objective is to improve program slicing, such that it becomes a feasible tool for debugging and its many other applications, for contemporary programming languages. Features like exception handling, complex data structures, concurrency or even `return` statements are all commonplace in programming languages used by the majority of programmers, but program slicing does not have complete solutions to the challenges they pose.

This goal, desirable as it might be, cannot be achieved by a single thesis project. Thus, we have limited the scope of the thesis to the four aforementioned features. We focused our work on the SDG and its related graphs, improving existing solutions or disproving their efficacy, to then propose a correct solution, proving it either formally or empirically.

Exception handling

We propose a new system to handle exception-related constructs such as *try*, *catch* and *throw*, based on Allen and Horwitz's [4]. We introduce conditional control dependence, a new kind of control dependence necessary to properly represent the effect of *catch* statements on the execution of a program, and the effect of the inclusion or exclusion of such statements in slices. We define an extension of the SDG, the exception-sensitive SDG (ES-SDG) and describe the process to create it and the algorithm to produce slices from it.

The contributions are: (1) a counter-example to Allen and Horwitz's technique [4], showing that it systematically excludes *catch* statements; (2) a new dependence, conditional control dependence, specifically designed for *catch* statements; (3) the ES-SDG, a graph combining [4] and conditional control dependence, and a matching slicing algorithm; (4) a theorem of completeness with its corresponding proof; (5) an implementation of the technique in two different slicers (*JavaSlicer* and *e-Knife*); and (6) an empirical evaluation of *JavaSlicer*'s implementation of the ES-SDG, comparing it to Allen and Horwitz's technique.

Unconditional jumps

Instructions like `continue`, `return`, `goto` or `throw` all require special representation in the SDG, as they alter the normal flow of the program in a way that the classic SDG is not equipped to handle. Solutions to this problem exist already, with the two most important ones being the augmented PDG or APDG [8] and its more precise successor, the pseudo-predicate PDG or PPDG [62]. We explore the situations in which the PPDG is not suitable (such as in the presence of `gotos` [49]), and the APDG must be used instead. Additionally, in cases where the PPDG might be suitable, we show additional restrictions that must be followed when a slicing technique is combined with the PPDG.

The contributions are: (1) the limitations of the PPDG, (2) the requirements that must hold for the PPDG slicing algorithm to work correctly, and (3) how those requirements are not met in different slicing techniques, such as the SDG, slicing object-oriented programs [34] and exception handling. For each of the three cases, we showcase a counter-example and propose a solution such that the technique can meet the aforementioned requirements.

Concurrency

Our work on concurrency is split in two areas. The first is related to the representation of *interference dependence* (which connect variables defined and used in different threads), and the second is the representation of synchronizations in a modelling language.

Regarding interference dependence, we analyse how a previous approach to this problem by Nanda and Ramesh [79] did not correctly account for the calling context, producing overly large slices. We propose a new technique, called the threaded SDG or tSDG. The tSDG mirrors the techniques used to make SDG precise in interprocedural scenarios. The result are slices with a level of precision that was not possible without an algorithm with exponential cost.

Our contributions are: (1) a counter-example for Nanda and Ramesh's technique [79], and the reasoning why it did not achieve the precision they expected, (2) a new technique for slicing shared-memory interprocedural programs precisely, in the form of a graph (the tSDG) and its slicing algorithm (how to traverse it to produce slices).

We have also studied the CSP modelling programming language[2], to

[2]CSP stands for communicating sequential processes. It is used to simulate complex concurrent systems and prove properties of such systems. CSP programs are therefore commonly referred to as specifications, as they specify the characteristics of programs in other languages.

model synchronizations in concurrent programs. We have defined a causally-consistent system to explore the execution of a CSP specification, meaning that the system only allows the exploration of steps that are the result of a possible interleaving of threads.

Our contributions are: (1) an extension of CSP tracks to include global timestamps, called R-tracks; (2) definitions to run through an R-track, with the optional restriction of causally-consistent steps; (3) theorems and proofs on the deterministic elements of this process; (4) an implementation of this technique called *reverCSP*, written in Erlang; and (5) an empirical evaluation of the implementation, showing the memory and time overhead and the size of the tracks generated for a given set of CSP benchmarks.

Complex data structures

As is the case with concurrency, our work on complex data structures is split. On the one hand, we extend our own previous work on field-sensitive program slicing (i.e., slicing that is still precise when data is being packed and unpacked in complex data structures) to adapt it to interprocedural programs, and on the other hand, we discuss the undecidability of various data analyses for programs with varying feature sets.

Our work on interprocedural field-sensitive program slicing successfully merges our field-sensitive technique [37] with an interprocedural slicing strategy called *tabular slicing* (an alternative to the SDG), which was originally defined for dataflow analysis [94]. We describe the combined graph and slicing algorithm, and empirically measure the overhead added by merging the two techniques.

Our contributions are: (1) the combination of field-sensitive slicing and tabular slicing to produce a precise, interprocedural field-sensitive slicer; (2) the implementation of this new technique in *e-Knife*, an Erlang slicer; and (3) an empirical evaluation of the slicer, both using slicing-specific benchmarks and real-world, widely-used Erlang libraries.

Although undecidability of data analysis might seem unrelated to complex data structures, we have studied the requirements for certain classes of data analyses to be undecidable, and we show that complex data structures is one of the main requirements to produce undecidability. For programs that are terminating, we show how three different data analyses are undecidable in the presence of complex data structures, loops (or `gotos`) and branching instructions. To do this, we extend the work by Reps [93], who stated that data dependence analysis for programs with functions, calls, and complex data structures is undecidable in the general case. Like him, we base our findings in comparing dataflow analyses and data dependence analyses to the Post's

Parenthesis Correspondence Problem (P-PCP), a well-known undecidable problem.

Our contributions are: (1) a classification of data analyses and a re-classification of Reps' results; and (2) an extension to Reps' counter-example, removing functions and calls to show that complex data structures are sufficient to produce undecidability on data analyses.

1.4 Structure of this thesis

Like most program slicers, this thesis splits its contributions into improvements to control and data dependence. There are five parts: an introduction, control dependence, data dependence, software tools, and conclusions.

I. The first part contains two chapters: Chapter 1, an introduction that motivates the rest of the document, a brief explanation of program slicing and the contributions of this work; and Chapter 2, which gives a brief background on shared concepts, like slices, common metrics, a categorization of static analyses, and the classic process used to produce slices: the SDG.

II. Control dependence represents the relationships between statements when a statement controls whether another executes, or the number of times it does. In this part, we explore three aspects of control dependence that can be improved upon:

- Chapter 3 [30, 33, 35] explores the effect of exception-handling in slicing, specially *catch* statements. To correctly represent them in the SDG, it introduces a new kind of control dependence: *conditional control dependence*. Then, it transforms the slicing graphs described in the background into exception-sensitive graphs, which produce better slices in the presence of exceptions. A new slicing algorithm for the modified graph is also described, along with an empirical evaluation and a description of alternative techniques to slice exception-laden programs.
- Chapter 4 [29, 31, 32] analyses the problems posed by unconditional jumps (e.g., `break` and `return`), and how they have been historically included in the SDG. Two alternatives are explored, and then we explore the problems and issues that arise when we apply these techniques in combination with other improvements to program slicing, such as object-oriented programs, exceptions, and even function calls.
- Chapter 5 [39] delves into the ever-complicated subject of concurrency. Through the CSP programming language, often used to

model concurrency, we explore the debugging of synchronizations in concurrent programs, and establish a system to ensure that the debugger can never produce an impossible interleaving of the processes. The result is a debugger that can travel backwards in time, always maintaining causal consistency.

III. Data dependence encompasses the links between statements based on data flow or variable assignments and usage. This part is split into three chapters, each tackling a different aspect of data dependence:
   - Chapter 6 [36, 40] presents a technique for slicing of shared-memory concurrent programs, specifically for the representations of interference, or the link between a definition and a usage of a shared variable across threads. Our representation of interference is precise even when it crosses the function boundary, i.e., it is context-sensitive.
   - Chapter 7 [28] tackles another classic software analysis conundrum: undecidability. Specifically, it explores which data analyses are undecidable, and for which group of programs. It also reflects on the axes of control-sensitivity and field-sensitivity, or the precise analysis in the presence of function calls and complex data structures, respectively.
   - Chapter 8 extends our previous work on field-sensitive program slicing [37], which can precisely slice complex data structures, and combines it with tabular slicing, an existing context-sensitive slicing technique, to produce a context- and field-sensitive slicer.

IV. Tools & Software: this thesis has enabled the creation or improvement of multiple software tools, implementing the algorithms, techniques, and ideas described in parts II and III. The three main tools are *JavaSlicer* [26] (Chapter 9), *e-Knife* (Chapter 10) and *reverCSP* [38] (Chapter 11). The first two are program slicers for Java and Erlang, respectively, and are implemented in Java. The last is a CSP runner/debugger with the capability of moving back and forth in time, maintaining causal consistency. Each tool's chapter describes it, shows the basic usage with practical examples and describes its implementation, such that they can be used and extended by other researchers.

V. Finally, the conclusions are laid out on Chapter 12 and future work avenues and open lines of research can be found on Chapter 13.

The end of this thesis contains two appendices. Appendix A lists the publications that make up this thesis and Appendix B proves Theorem 3.1 from Chapter 3. The proof was split from the chapter due to its length relative to the chapter.

Chapter 2

Preliminary concepts and definitions

To keep this thesis self-contained, we first need to explain and define some key concepts in program slicing. Concepts and definitions from this chapter will be used in other chapters, although some will include a small background section, in cases where the definitions are only related to a single chapter. Readers already familiar with program slicing as an analysis technique and the System Dependence Graph as a data structure used for slicing may want to skip this chapter and only use it as reference material.

2.1 The basics

Program slicing is an analysis that typically has two inputs (a program to be analysed and a criterion to analyse it with) and an output: the slice. There are many different kinds of slicing techniques, but this work focuses mostly on backwards static slicing, which answers the question: *What elements of a program can affect (in some execution) a given point in the program?* The point is often referred to as the slicing criterion, and can be defined as:

Definition 2.1 (Slicing criterion)**.** *Given a program P, a* slicing criterion $sc = \langle p, \mathcal{V} \rangle$ *is a tuple, where p is a position in P and $\mathcal{V}$ is a finite set of variables in the scope of p.*

The position in a program will be typically indicated with a line number, assuming that programs are formatted with one instruction per line. When a slicing criterion only contains a single variable, the set notation will be omitted. For example, $\langle 10, x \rangle$ represents a slicing criterion in which the position is the instruction located in line 10 and $\mathcal{V} = \{x\}$.

To answer the question posed by static backward slicing correctly, we have to formalize the meaning of *affect*. Informally, we understand that some

instructions or statements in a program affect others. For example, `x = 1` affects instructions after it that use the value of `x`, whereas `print(x)` does not. Formally, we will compare the values of the variables in *sc* in all executions of the original program against the same executions (with the same inputs) in the sliced program. To compare executions, we define a sequence of values assigned to each of the variables in $\mathcal{V}$ each time the criterion (p) is executed:

Definition 2.2 (Sequence of values)**.** *Let P be a program, let $sc = \langle p, \mathcal{V} \rangle$ be a slicing criterion and I a set of input values for P. A* sequence of values $\text{SEQ}(P, sc, I) = [e_0, e_1, ...]$*, is a list of variable-value assignments, where e_i contains the value u of each variable v when the execution of P with input I reached p for the i-th time ($e_i = \{v \leftarrow u_i \mid v \in \mathcal{V}\}$).*

> **Example 2.1** (Sequences of values in Fig. 1.1)**.**
> Let P and S be the program and its corresponding slice shown in Fig. 1.1. The sequence focused on the slicing criterion $\langle 10, \texttt{sum} \rangle$ when the program reads the value 10 for `n` would be represented as $\text{SEQ}(P, \langle 10, \texttt{sum} \rangle, \{\texttt{n} \leftarrow 10\})$. Its value would be $[\{\texttt{sum} \leftarrow 55\}]$, as line 10 is executed once, and the value of `sum` is 55 ($\sum_{i=0}^{10} i$). Because the slice of P is a correct slice, the sequences $\text{SEQ}(P, \langle 10, \texttt{sum} \rangle, \{\texttt{n} \leftarrow 10\})$ and $\text{SEQ}(S, \langle 10, \texttt{sum} \rangle, \{\texttt{n} \leftarrow 10\})$ match. Moreover, as shown in the next definition, all sequences of S and P with the same slicing criterion will match (for any possible input).

The slicing criterion in the sequence of values normally contains a single variable, but can contain multiple, even variables that do not explicitly appear in the given line, as long as that program location is in their scope. However, we can also define criteria with no variables. In that case, the slice only tries to match the number of executions of the criterion's line, as shown in the next example.

> **Example 2.2** (Sequence of values with "empty" criteria)**.**
> The sequence of values can result in simpler or more complicated data structures depending on the size of $\mathcal{V}$ in a given slicing criterion, as well as the number of times that the slicing criterion is executed. For example, $\text{SEQ}(P, \langle 2, \emptyset \rangle, I) = [\{\}]$ when P executes line 2 once, and $[\{\}, \{\}]$ when it does so twice.

If we compare programs with respect to the sequence of values they can produce, we can define one as the slice of another:

Definition 2.3 (Executable backwards static slice)**.** *Let P be a program, and $sc = \langle p, \mathcal{V} \rangle$ be a slicing criterion. P' is an* executable backwards static slice *of P ($P' = \text{SLICE}(P, sc)$) if:*

1. *P' is a well-formed executable program,*
2. *$P' \subseteq P$ (P' is the result of removing zero or more statements from P), and*
3. *for all possible inputs I, $\text{SEQ}(P, \langle p, \mathcal{V} \rangle, I)$ is a prefix of $\text{SEQ}(P', \langle p', \mathcal{V} \rangle, I)$, where p' is a position in P' that corresponds to p in P.*

The three adjectives that qualify this definition show that there are many different types of slices. Slices can be executable or not, the direction of the slice changes whether we want the statements that *affect* the criterion (backwards) or those that are *affected* by it (forwards), and we can be interested in all possible inputs for the program (static), or in just a single execution (dynamic). Most of the techniques proposed in this document are valid for forwards or dynamic slicing, but they will be demonstrated, and proven theoretically and empirically for this definition of slice.

A final variation that will appear during the course of this work is amorphous versus *syntax-preserving* slices. The latter must be the result of removing statements from the original program, whereas the former are not bound by this condition. Definition 2.3 indicates that slices are *syntax-preserving* due to condition 2, but this will be relaxed to include amorphous slices in some chapters.

2.2 Measuring and comparing slices

For slices to be useful, it is not enough for them to meet the conditions established in Definition 2.3, because according to it, any program is a slice of itself w.r.t. any possible slicing criteria. There are two dimensions in which all slices are measured:

Precision How many of the instructions in the slice could really affect the slicing criterion? This is typically measured with the ratio of instructions necessary for the slice vs. instructions included in the slice.

Recall How many of the instructions that could affect the slicing criterion are included in the slice? This can be expressed as the percentage of instructions included in the slice w.r.t. instructions that should have been included. If recall is not perfect, Definition 2.3 does not hold (due to condition 3), and thus the output cannot be called a slice. We will refer to them as *incomplete slices*.

In the literature of program slicing, precision and recall are often referred to as *correctness* and *completeness*, respectively. Some of the most important errors to capture in the slicing process are completeness errors, which would make the output of the process not a slice, and thus not useful for many of the applications of slicing. On the other hand, achieving perfect precision and

```
1  read(n);
2  sum = 0;
3  prod = 1;
4  i = 0;
5  while (i < n) {
6    sum  += i;
7    prod *= i;
8    i++;
9  }
10 write(sum);
11 write(prod);
```

(A) An imprecise or incorrect slice.

```
1  read(n);
2  sum = 0;
3  prod = 1;
4  i = 0;
5  while (i < n) {
6    sum  += i;
7    prod *= i;
8    i++;
9  }
10 write(sum);
11 write(prod);
```

(B) An incomplete slice.

FIGURE 2.1: Two imperfect slices of the code in Fig. 1.1a w.r.t. $\langle 10, sum \rangle$.

recall means producing the most efficient slice, and thus it has a specialized name, the minimal slice:

Definition 2.4 (Minimal slice)**.** *A slice* $S = \text{SLICE}(P, sc)$ *is* minimal *if there does not exist any other slice* $S' = \text{SLICE}(P, sc)$ *such that* S' *has fewer statements than* S*.*

Example 2.3 (Incorrect, incomplete, and minimal slices)**.**
The code shown on Fig. 2.1a is an imprecise or incorrect slice of the code in Fig. 1.1a. Specifically, it includes line 7, but that line cannot affect the slicing criterion $\langle 10, sum \rangle$ in any execution. However, the code is still a valid slice, as it does not exclude any necessary instruction.

Conversely, Fig. 2.1b shows an incomplete slice, due to the exclusion of line 8 (`i++`). This statement is necessary to compute the correct value of *sum* at line 10 in two ways: *i* affects both the number of times that the loop is executed, and the value that is added to *sum*. In turn, the loop affects how many times *i* is added to *sum*, altering the value at the slicing criterion.

Finally, the minimal slice appears on Fig. 1.1b: there is no slice smaller that can still compute the same values as the original program.

In simple examples, valid and even minimal slices can be computed by readers at a glance. However, given an arbitrary program, computing its minimal slice is undecidable, and most program slicing techniques (with ORBS [10] being a notable exception) are unable to detect misdirections such as the one shown on Fig. 2.2. Most slicers will output Fig. 2.2b, not realizing that lines 2 and 3 cancel each other and thus can be removed, but only when both are removed (as in Fig. 2.2c). Additionally, a program might have multiple

```
1 read(x);
2 x++;
3 x--;
4 x -= 1;
5 print(x);
```

(A) A program

```
read(x);
x++;
x--;
x -= 1;
print(x);
```

(B) An imprecise slice.

```
read(x);
x++;
x--;
x -= 1;
print(x);
```

(C) Minimal slice 1.

```
read(x);
x++;
x--;
x -= 1;
print(x);
```

(D) Minimal slice 2.

FIGURE 2.2: A program with a redundant pair of increment/decrement instructions.

minimal slices (Figs. 2.2c and 2.2d), which makes testing program slicers a specially tricky subject.

2.3 A sense on sensitivity

In the field of static analysis, a specific analysis is said to be context-sensitive or field-sensitive [74] (sometimes called *structure-transmitted* [92]) when it takes into account the calling context (for the former) and elements of a complex data structure (for the latter), such as fields in objects or elements in tuples [102]. There are other sensitivities defined (e.g., flow-sensitivity) but they are not a relevant distinction in this work.

Program slicing is no different, and depending on how the slices are produced, we can say that slicers are (or not) context-sensitive when they are precise enough to exclude unnecessary calls to procedures that are included, and field-sensitive when they can track a value being packed into a larger data structure (e.g., a field in an object) and later unpacked. However, these features tend to come with a cost, as implementing them naively involves using stacks to track the current call stack or location of variables in data structures.

The following examples use data dependence graphs (DDG)[1] to showcase the difference between context- and field-sensitive analyses from their insensitive counterparts.

Example 2.4 (Comparing context-sensitivity and context-insensitivity)**.** Consider the Java code in Fig. 2.3, which contains two questions, Q1 and Q2: "Does y in this line depend on $(3, x)$?". Q1 and Q2 can be answered correctly with a context-sensitive DDG such as the one shown in Fig. 2.4.

[1] DDG represents the flow of data through a program. Section 2.4 defines flow dependence, a subset of data dependence, with Definition 2.8. The DDG is the graph representation of that definition.

```
1  int x, y;
2  void main() {
3    x = 42; // A unique value in the program
4    g();
5    // Q1: Does (5,y) depend on (3,x)?
6    x = 0;
7    g();
8    // Q2: Does (8,y) depend on (3,x)?
9  }
10 void g() {
11   y = x;
12 }
```

FIGURE 2.3: An interprocedural Java program.

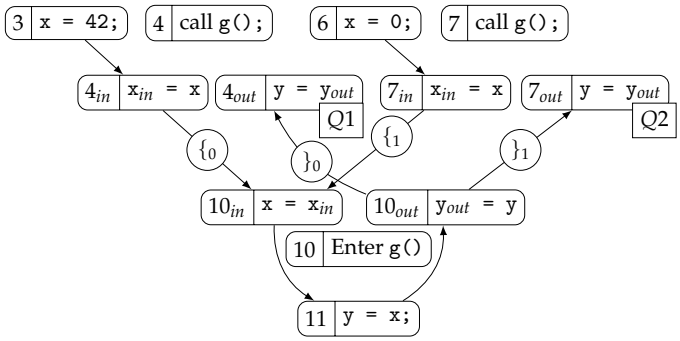


FIGURE 2.4: DDG of the code in Fig. 2.3. Input and output arcs are labelled with $\{_i$ and $\}_i$, respectively ($i \in \mathbb{N}$).

```
1    x = (0,[42,5]);  # 42 is a unique value
2    (a,b) = x;
3    [c,d] = b;
4    # Q1: Does (3,c) depend on 42?
5    # Q2: Does (3,d) depend on 42?
```

FIGURE 2.5: An intraprocedural Python program with composite data structures.

Observe that input and output data arcs are labelled in such a way that we can distinguish between the two calls to g.

In particular, we can answer question Q2 by traversing backwards the data arcs from Q2: $7_{out} \xleftarrow{\}_1} 10_{out} \leftarrow 11 \leftarrow 10_{in} \xleftarrow{\{_1} 7_{in} \leftarrow 6$. The first arc ($7_{out} \xleftarrow{\}_1} 10_{out}$) is labelled, so when 10_{in} is reached, we can only exit g by traversing the arc with the matching label ($10_{in} \xleftarrow{\{_1} 7_{in}$), and not the other one ($10_{in} \xleftarrow{\{_0} 4_{in}$). The traversal does not reach 3 and thus the answer to Q2 is *no*.

The same happens when we consider $Q1$: we start from node 4_{out}, and by traversing the arc with label $\}_0$ we cannot traverse the non-matching arc labelled $\{_1$. The traversal would be $4_{out} \xleftarrow{\}_0} 10_{out} \leftarrow 11 \leftarrow 10_{in} \xleftarrow{\{_0} 4_{in} \leftarrow 3$, which contains 3, so the answer to $Q1$ is *yes*.

The labels, therefore, allow us to distinguish between different calls to the same function. Another example of context-sensitive analysis that would solve this problem is the use of the *system dependence graph* [54], which incorporates a two-pass traversal and summary arcs to perform an equivalent analysis.

Context-insensitive analyses usually produce an over-approximation. For instance, in Fig. 2.4, if we ignore the labels in the arcs, statement 3 is reachable from question Q2 ($7_{out} \leftarrow 10_{out} \leftarrow 11 \leftarrow 10_{in} \leftarrow 4_{in} \leftarrow 3$). Therefore, this analysis would incorrectly conclude that variable y at Q2 depends on variable $(3, x)$ (it would answer *yes* to Q2).

Example 2.5 (Comparing field-sensitivity and field-insensitivity)**.**
Consider the Python code in Fig. 2.5, which contains two questions Q1 and Q2. Correctly answering these questions implies deciding whether the value of `c` and `d` depend on 42 at the end of the program, respectively.

These questions can be answered correctly with a field-sensitive analysis such as the one shown in Fig. 2.6. Observe that data structures are tokenized, and the arcs between them are labelled with the type of data structure and

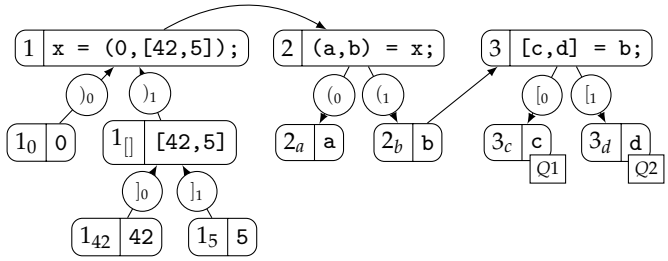


FIGURE 2.6: DDG of the code in Fig. 2.5. Arcs that represents an access to a data structure are labelled with $(_i$, $[_i$ (writes), or $)_i$, $]_i$ (reads). The kind of symbol used distinguishes lists '[]' from tuples '()', and $i \in \mathbb{N}$.

their position. For instance, $[_0$ means that the first element of a list is required; and $(_1$ means that the second element of a tuple is required. When the closing labels ('$)_i$', '$]_j$') are found, they can only be traversed if the last unfulfilled requirement ('$(_a$', '$[_b$') matches it. For example, the sequence $2_a \xleftarrow{(_0} 2 \leftarrow 1 \xleftarrow{)_1} 1_{[]}$ would not be valid.

To answer question $Q1$, we can traverse the graph backwards from 3_c. Following the sequence $3_c \xleftarrow{[_0} 3 \leftarrow 2_b \xleftarrow{(_1} 2 \leftarrow 1 \xleftarrow{)_1} 1_{[]} \xleftarrow{]_0} 1_{42}$, we would reach 42, and thus `c` could actually be 42: $Q1$'s answer is *yes*. Regarding $Q2$, we start the traversal at 3_d, and obtain the sequence $3_d \xleftarrow{[_1} 3 \leftarrow 2_b \xleftarrow{(_1} 2 \leftarrow 1 \xleftarrow{)_1} 1_{[]} \xleftarrow{]_1} 1_5$. Because there does not exist a correct path from 3_d to 42, the answer to $Q2$ is *no*.

If the labels in Fig. 2.6 were ignored, the DDG would become field-insensitive. The traversal from $Q2$ would be able to produce the sequence that reaches 42 ($3_d \leftarrow 3 \leftarrow 2_b \leftarrow 2 \leftarrow 1 \leftarrow 1_{[]} \leftarrow 1_{42}$), and the answer to $Q2$ would incorrectly be *yes*.

Although the example uses labels, there are other field-sensitive data analysis techniques, such as pre-processing the code in order to only read or write to a single element of a data structure in each statement (`{a, b} = x` would become `a = x_0; b = x_1`). This is known as *atomization* (see [91] and [78, Ch. 12.2]), which solves the problem by splitting complex assignments into multiple ones, as illustrated in the following example. In turn, the graph does not mix together different parts of a variable in a single node.

```
1  x_0   =  0
2  x_1_0 = 42
3  x_1_1 =  5
4  a = x_0        # Depends on 1
5  b_0 = x_1_0    # Depends on 2
6  b_1 = x_1_1    # Depends on 3
7  c = b_0        # Depends on 6
8  d = b_1        # Depends on 7
```

FIGURE 2.7: An atomized version of the program in Fig. 2.5.

Example 2.6 (Field-sensitive slicing with atomization)**.**
Consider the code in Fig. 2.5, and its atomized version in Fig. 2.7. Each assignment that contains a complex data structure (explicitly or implicitly) is split into several assignments using indices. For an example of explicit atomization, the first line in the original program is split into lines 1-3 of the atomized program. Lines 5-6 represent a structure that is not present in the source code (implicit), but reflect the internal structure of the second element of `x` (and `b`). In this case, atomization can solve both Q1 and Q2. For Q1, we start at line 8, which depends on line 6, which in turn depends on 2, so we can answer positively. Onto Q2: starting at line 9, it depends on 7, which depends on 3. Thus, the answer to Q2 is no, and both questions have been answered correctly.

2.4 Graphs to compute slices

There are various techniques that can be used to produce slices of a given program, but the most popular one revolves around dependence graphs: the program's statements are represented as vertices or nodes in the graph, and the relationships between them (which we will call dependences) are represented as directed edges or arcs. Once this representation is built, producing slices is a reachability problem through the graph, which can be solved in linear time. However, building dependence graphs is slightly more involved.

The two most popular dependence graphs are the *program dependence graph* (PDG) [24], and the *system dependence graph* (SDG) [52], which builds upon the PDG, incorporating context-sensitivity. This section explains the process of creating a SDG for simple, imperative programs, and we will expand on this process on Chapters 3, 4, 6 and 8.

The SDG is built upon the PDG and, in turn, the PDG is computed w.r.t. the *control-flow graph* (CFG) [3, 44], a commonly used graph to represent the

execution flow of a program. To define these graphs, we will use the * suffix operator to signify the reflexive and transitive closure of a set of arcs (e.g., A^* is the reflexive and transitive closure of A). In the CFG, nodes represent instructions, except for two nodes that represent the *Enter* (or Start) and *Exit* (or End) of the graph. A node n is connected with a directed arc to a node n' if there exists an execution where n' is executed immediately after n (according to the programming language's semantics). Formally,

Definition 2.5 (Control-flow graph [3])**.** *Given a procedure P, its* control-flow graph *(CFG) is a directed graph $G = (N, A)$, where N is a set of nodes that contains a node for each instruction in P, plus two nodes named* Enter *and* Exit*; and A is a set of directed arcs $A \subset N \times N$ connecting the nodes, in which these conditions must always hold:*

1. $\forall n, n' \in N \,.\, (n, n') \in A \iff$ *n' can be executed immediately after n (it is assumed that* Enter *is executed immediately before the first instruction of the procedure, and* Exit *immediately after the last instruction(s)),*
2. $\forall n \in N \,.\, (Enter, n) \in A^*$ *(all nodes can be reached from* Enter*), and*
3. $\forall n \in N \,.\, (n, Exit) \in A^*$ *(*Exit *can be reached from any node).*

To avoid dealing with malformed programs, conditions 2 and 3 guarantee that there are no infinite loops and no unreachable code (which is a compilation error in some, but not all programming languages). It is important to note, however, that traditionally, the CFG (and thus, static program slicing) has not evaluated expressions in the source code. Therefore, some infinite loops (like `while (true)`) are represented in the CFG as if the condition could evaluate to either `true` or `false`, instead of representing it as the infinite loop that it is (barring any jumps inside its body).

Depending on the number of outgoing arcs, we can classify nodes in two categories: *statements* have one outgoing arc (e.g., assignments, calls, variable declarations), whereas *predicates* have two or more outgoing arcs (e.g., loops and conditionals).

The main two dependences commonly found in imperative programs are control dependence and data dependence. The first contains the relationships between statements that affect whether the other executes or the number of times it is executed. The latter reflects interactions that relate to data, such as when variables are used and defined (flow dependence) or the evaluation of expressions and the path they take in larger instructions (value dependence).

Formally, control dependence is defined based on the structure of the CFG and using the concept of post-dominance:

Definition 2.6 (Post-dominance [104])**.** *Let $G = (N, A)$ be a CFG, and $n_1, n_2 \in N$ be two nodes.* n_2 post-dominates *n_1 in G if n_2 appears in every path between n_1 and* Exit*.*

Definition 2.7 (Control dependence [52]). *Let $G = (N, A)$ be a CFG and $n_1, n_2 \in N$ be two nodes. n_2 is* control dependent *on n_1 ($n_1 \to n_2$) if n_2 post-dominates one but not all of n_1's successors ($\{n \mid (n_1, n) \in A\}$).*

As aforementioned, data dependence is an umbrella term that contains different dependences related to data, such as flow dependence, which is based on definitions and usages of variables, as well as the paths formed by the CFG:

Definition 2.8 (Flow dependence [52]). *Let $G = (N, A)$ be a CFG and $n_1, n_2 \in N$ be two nodes. n_2 is* flow dependent *on n_1 ($n_1 \to n_2$) if (1) n_1 defines a variable v, (2) n_2 uses v, and (3) there is a path in G between n_1 and n_2 in which v is not defined.*

Although it is not often mentioned, a similar pattern appears between declarations and definitions (for languages with explicit declaration of variables). This dependence is often ignored or bundled onto flow dependence.

Definition 2.9 (Declaration dependence). *Let $G = (N, A)$ be a CFG and $n_1, n_2 \in N$ be two nodes. n_2 is* declaration dependent *on n_1 ($n_1 \to n_2$) if n_1 declares a variable v, and n_2 defines v.*

Control and flow dependence are the two main ingredients to build the PDG, which shares nodes with the CFG, but connects them with dependences instead of control-flow arcs:

Definition 2.10 (Program dependence graph [24]). *Given a procedure P and its CFG $G = (N, A)$, the* program dependence graph *(PDG) of P is $G' = (N', A')$, where $N' = N \setminus \{Exit\}$, $A' \subseteq N' \times N'$, and $(n, n') \in A'$ if and only if n' is control or flow dependent on n.*

Algorithm 1 Backward slicing algorithm with the PDG

```
1: function PDGSLICE(P, sc = ⟨p, V⟩)
2:     G = (N, A) ← BUILDPDG(P)
3:     n ← LOCATESC(N, p)
4:     S ← {n' | (n', n) ∈ A*}
5:     return PRUNEPROGRAM(P, S)
```

In the PDG, the two types of dependences are placed into a single set of arcs, but we will sometimes split them into two subsets: $A' = A_c \cup A_f$ (control and flow). From the PDG we can obtain our first slices. In Algorithm 1 we introduce the first (and simplest) slicing algorithm for dependence graphs: locate the node that represents the slicing criterion, traverse all arcs backwards

collecting the set of nodes reached, and prune the original program given that set of nodes to produce the sliced program.

This algorithm is the basis for dependence graph-generated slices, which is the main focus of this work. However, the PDG is defined around a single procedure. To precisely slice multi-procedure programs, the SDG was defined as an extension of the PDG, with three main changes: new nodes to connect calls, new arcs to summarise the body of procedures and a new slicing algorithm much more precise than the PDG's but with the same time complexity.

The first step is the creation of parameter and argument nodes: around each *Enter* node, *formal nodes* are created to represent the input of values to each parameter (and its output in the case of call-by-reference programming languages) and the output of functions (the returned value). These nodes will be referred to as formal-in for parameter input and formal-out for parameter and function output, and they will be labelled accordingly: formal-in with $\texttt{x} = \texttt{x}_{in}$ (the value comes into the function and is assigned to the parameter) and formal-out with $\texttt{x}_{out} = \texttt{x}$ (the value of the variable is passed back to the caller), where `x` is the name of the parameter. Equivalently, each call to a procedure produces additional nodes: one to represent the call itself, independent of the instruction that contains it (e.g. `a = b();` would produce a separate `call b()` node), and a set of *actual nodes* matching the set of formal nodes generated at the callee's *Enter* node. Actual nodes are labelled similarly to formal nodes: actual-in as $\texttt{x}_{in} = \texttt{a}$ and actual-out as $\texttt{a} = \texttt{x}_{out}$, where `x` is the parameter and `a` is the argument passed.

The newly added nodes are connected to the rest of the graph as follows: each formal node is control dependent on the *Enter* node, each call node is control dependent on the instruction that contains that call, and each actual node is control dependent on its call node. Additionally, each actual-in node is connected to its matching formal-in node with an arc (which we call *input arc*, and its set is A_{in}), each call to its matching *Enter* node (with a *call arc*, collected at A_{call}), and each formal-out to each of its matching actual-out nodes (with an *output arc*, with set A_{out}).

The second step is the computation of summary arcs, which is one of the more computationally intensive steps to build a SDG. Summary arcs are computed by computing slices starting at each formal-out node. For the computation of these slices, input, call, and output arcs are ignored. After each slice is completed, each formal-in reached generates a set of summary arcs: if a slice starting from formal-out b included formal-in a, then the following summary arcs would be generated:

$$A_{sum} = \left\{(a', b') \mid (a', a) \in A_{in} \wedge (b, b') \in A_{out} \wedge (n, a'), (n, b') \in A_c\right\}$$

Algorithm 2 Backward slicing algorithm with the SDG [53]

```
1: function SDGSLICE(P, sc = ⟨p, V⟩)
2:     G = (N, A) ← BUILDSDG(P)
3:     n ← LOCATESC(N, p)
4:     S1 ← {n' | (n', n) ∈ A* \ A_out}                    ▷ 1st pass.
5:     S2 ← {n'' | n' ∈ S1, (n'', n') ∈ A* \ (A_in ∪ A_call)}   ▷ 2nd pass.
6:     return PRUNEPROGRAM(P, S2)
```

which includes all pairs of matching actual-in a' and actual-out b' that share a call (n). However, in the computation of summary arcs, newly added summary arcs must be taken into account, so the process of slicing w.r.t. each formal-out must be repeated until no new arcs are added. Now, we can formalise the SDG:

Definition 2.11 (System dependence graph [53])**.** *Let $P = \{P_1, P_2...P_n\}$ be a program with n procedures, where each P_i is a procedure; and let $G_i = (N_i, A_i)$ be the PDG of P_i. The* system dependence graph *of P is $G = (N, A)$, where*

- $N = \bigcup_{i=1}^{n} (N_i \cup \text{ACTUALNODES}(P_i) \cup \text{CALLNODES}(P_i) \cup \text{FORMALNODES}(P_i))$
- $A = A_{in} \cup A_{call} \cup A_{out} \cup A_{sum} \cup (\bigcup_{i=1}^{n} A_i)$

The third and final step is to define a new slicing algorithm that takes advantage of summary arcs to produce context-sensitive slices. Algorithm 2 is very similar to the PDG slicing algorithm, with a small twist: instead of traversing all arcs unconditionally, the algorithm uses a two-pass process. In the first pass, output arcs are ignored, so the traversal does not reach into calls, but does go up through the callers (via input and call arcs). The second pass continues the traversal from every node reached in the previous pass, but this time ignoring input and call arcs. This has the effect of going "down" through calls, but ignoring other callers of those calls, improving precision.

With the SDG slicing algorithm, we can produce precise, context-sensitive slices of imperative programs.

Part II

Control Dependence

Chapter 3

Exception handling and conditional control dependence

> When you *try* your best, but you don't succeed.
>
> *Coldplay, Fix You*

Exception handling constructs are an alternative to the use of `goto` and similar unstructured jumps to handle errors during the normal execution of a program. Nowadays, it is present in most imperative programming languages, with the notable exception of C and Go. However, program slicing still does not produce complete slices in the 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 [4], which in turn extended Sinha's proposal [100]. 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 3.1.

Example 3.1 (Incompleteness when slicing *try-catch* constructs in [4])**.** Consider the Java program shown in Fig. 3.1a, 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, see Example 2.2). The slice produced by Allen and Horwitz can be seen in Fig. 3.1b. It removes the whole `catch` block, and thus it is incomplete, as line 9 will only execute

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

(A) Original program

```
public void f() {
  try {
    g();
  } catch (E e) {}
  g();
}

public void g() {
  throw new E();
}
```

(B) Allen and Horwitz's slice

```
public void f() {
  try {
    g();
  } catch (E e) {}
  g();
}

public void g() {
  throw new E();
}
```

(C) The correct slice

FIGURE 3.1: Java program that throws two exceptions of type E, but captures only the first one.

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.

3.1 A new kind of dependence generated by *catch* statements

Example 3.1 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. 3.2, which shows that the *catch* statement is not part of

```
1 void main(int x) {
2   try {
3     throw new E();
4   } catch (E e) { }
5   log(x);
6 }
```

(A) The original program.

```
void main(int x) {
  try {
    throw new E();
  } catch (E e) { }
  log(x);
}
```

(B) The slice w.r.t. $\langle 3, \emptyset \rangle$.

```
void main(int x) {
  try {
    throw new E();
  } catch (E e) { }
  log(x);
}
```

(C) The slice w.r.t. $\langle 5, \emptyset \rangle$.

FIGURE 3.2: Slices of a program that throws and catches an exception.

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. 3.2b); or if the slicing criterion is located after the catch statement (see Fig. 3.2c). 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. 3.2b).
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. 3.2c).
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. 3.2a).

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 3.1 (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 are* 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.*

Conditional control dependence might seem counterintuitive, as it creates a three-way relationship, $s_1 \rightarrow \{s_2, s_3\}$, from a *catch*, s_1, to the source of an exception, s_2 (and a later statement, s_3). If we think of it as the *catch* controlling a *throw* statement, it does not make sense, as the real relationship is the inverse (a *catch*'s execution is controlled by the *throw*'s execution), but in this case the *throw* acts as a sort of catalyst for the dependence, i.e., $s_1 \rightarrow s_3$ if and only if s_2 is in the slice ($s_1 \xrightarrow{s_2} s_3$). To simplify the algorithm, we express it as a $s_1 \rightarrow \{s_2, s_3\}$ relationship.

3.2 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 2.4.

On top of the creation of the SDG, we need to consider the representation of unconditional jumps (e.g., *throw*), with non-executable control-flow (additional arcs in the CFG to complement control dependence), replacing the CFG with the ACFG and the PDG with the APDG[1] (in both, the leading A stands for *augmented*). A full explanation of the ACFG and APDG, is available on Section 4.1.

We organize our modifications considering the different graphs used to build a SDG: ACFG, APDG, and finally SDG. In the following, to clearly differentiate between each version of the graph, our extended graphs are prefixed with 'ES-', which stands for "exception-sensitive", so the ACFG becomes the ES-CFG, the APDG becomes the ES-PDG, and the SDG becomes the ES-SDG.

[1]The APDG can be replaced by its more precise counterpart, the PPDG (pseudo-predicate program dependence graph), described in Section 4.1.2, but only if the program being analysed has no *goto* statements.

3.2.1 Modifications to the ACFG to create the ES-CFG

In this section we compositionally describe how to construct any ES-CFG: 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-CFG 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.

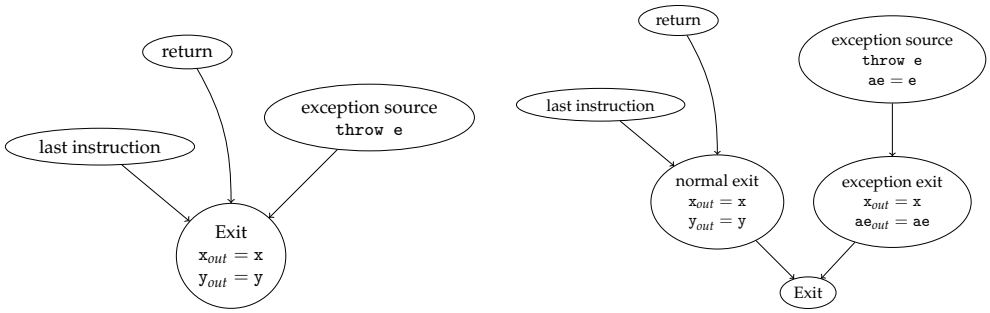
Procedure declarations with exceptions. This case represent those functions definitions that contain a potential source of exceptions, e.g., a *throw* statement, a possible division by zero, or a call to other procedure that may throw an exception. If the procedure contains exception sources, the *Exit* node contained in the original ACFG is split into three nodes: *normal exit*, *exception exit*, and *Exit* [4].

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. 3.3 shows how this exit-node transformation is done showing the difference between an ACFG and an ES-CFG 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.



(A) ACFG procedure declaration with exceptions

(B) ES-CFG procedure declaration with exceptions

FIGURE 3.3: The ES-CFG uses two exit nodes (*normal exit* and *exception exit*) to differentiate normal and abrupt termination of a function.

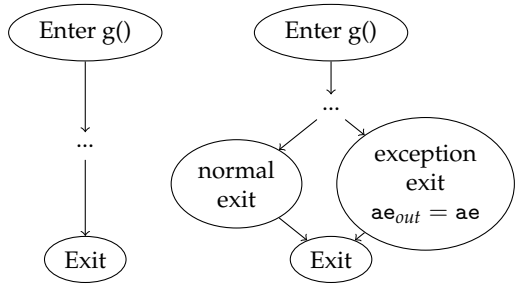


FIGURE 3.4: Partial ACFG (left) and ES-CFG (right) of the *Enter* and *Exit* of the program in Fig. 3.1.

Example 3.2 (The ES-CFG (Part 1): *Enter* and *Exit*)**.**
Consider again the code in Fig. 3.1. Figure 3.4 shows part of the g's CFG on the left, and its matching ES-CFG on the right. Because g contains a source of exceptions, its *Exit* node is split in two. The same transformation would be applied to `f`, even though the exception thrown is captured in all cases. This is because we need to model the absence of `f`'s *catch* statement, and in that case, the method could end with an exception. However, the ES-CFG of a procedure that calls `f` would not contain an *exception exit*, as no exception source appears (either caught or uncaught) in it.

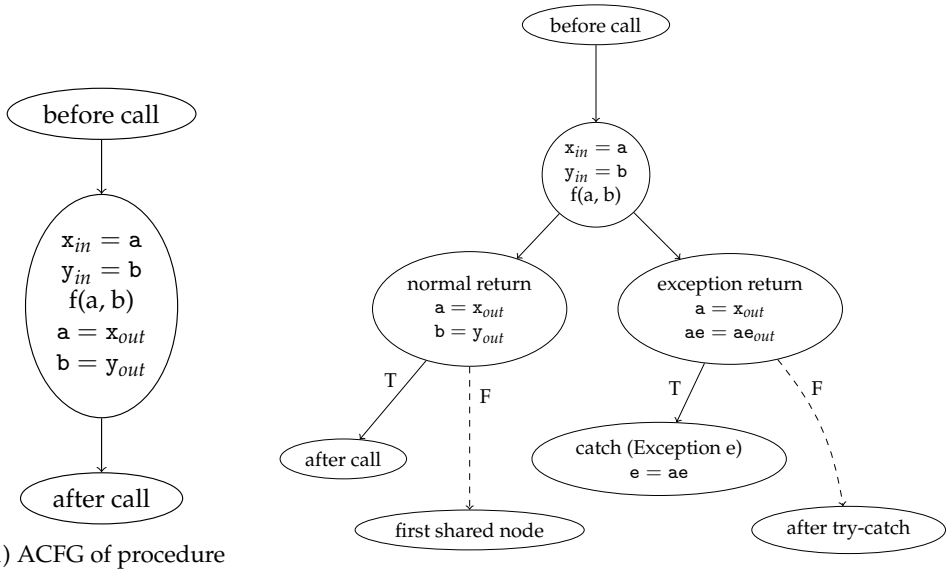
Calls to procedure definitions that may throw exceptions. These calls must be also redefined to differentiate whether the procedure ended with a normal execution (normal exit) or an exception was raised and uncaught during the execution (exception exit). The treatment is analogous to the one described 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 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. 3.5a shows the difference between the ACFG structure, where the behaviour is the same for both normal and exception procedure execution, and the ES-CFG structure which differentiates both possibilities resulting into different execution



(A) ACFG of procedure call with exceptions

(B) ES-CFG of procedure call with exceptions

FIGURE 3.5: The ES-CFG distinguishes between four paths to represent function calls that may produce exceptions.

paths. Note that both *return* nodes may not output the same variables, as some may have not been modified when the exception is thrown.

Example 3.3 (The ES-CFG (Part 2): calls)**.**
The program in Fig. 3.1 contains two calls to g. Figure 3.6 shows the transformation from ACFG to ES-CFG of both calls. The first one, inside the *try*, is very similar to the generic transformation shown on Fig. 3.5. However, the node after the call, the node after the `try-catch` and the first shared node happen to be the same one (the second call to g). The second call's representation differs more from the generic transformation. Because no *catch* is present to capture the exception present in *exception return*, it is connected to the *exception exit*, and its non-executable control-flow connects to *Exit* (again, because no *catch* is present). On the other hand, *normal return*'s next instruction would be *normal exit*, and the first node shared between the exception and normal paths is *Exit.*

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

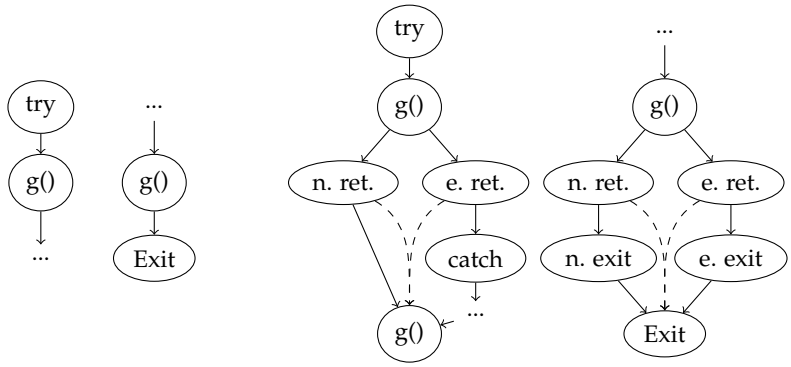
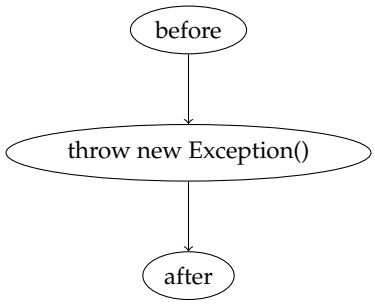
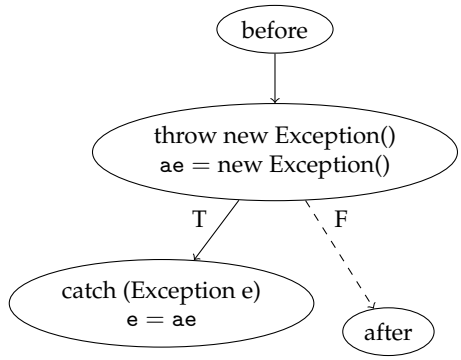


FIGURE 3.6: Partial ACFG (left) and ES-CFG (right) for calls in the program of Fig. 3.1.



(A) ACFG of an unconditional exception source

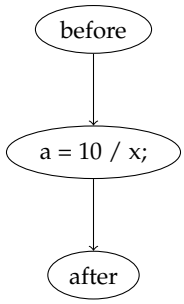


(B) ES-CFG of an unconditional exception source

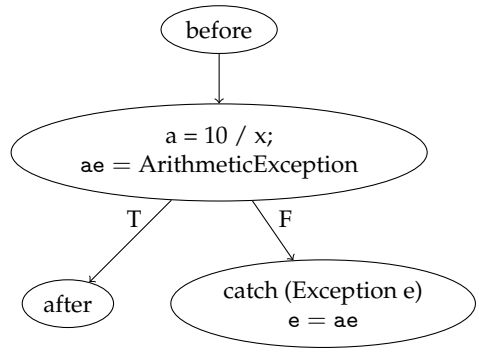
FIGURE 3.7: Representation of a `throw` statement in the ACFG and ES-CFG.

ES-CFG as pseudo-predicates [4]. 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. Figure 3.7 shows an example of how a `throw` instruction is represented both in the ACFG and in the ES-CFG.

Example 3.4 (The ES-CFG (Part 3): *throw*)**.**
The program in Fig. 3.1 has a single source of exceptions: a *throw* statement (unconditional source of exceptions). Thus, its representation will be similar



(A) ACFG of a conditional exception source



(B) ES-CFG of a conditional exception source

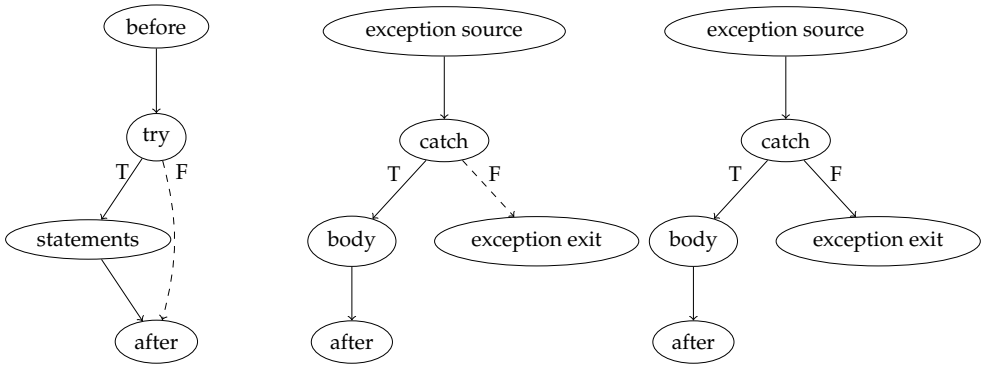
FIGURE 3.8: Representation of a conditional exception source in the ACFG and ES-CFG.

to the one shown on Fig. 3.7b, with its executable control-flow leading to the *exception exit* (as there is no *catch* in procedure g). Its non-executable control-flow will connect it to *normal exit*, as there is no instruction afterwards.

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. Figure 3.8 shows an example, displaying the difference between the ACFG single path representation and the ES-CFG 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-CFG 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 [8]. 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



(A) `try` instruction (B) `catch` pseudo-predicate (C) `catch` predicate

FIGURE 3.9: Representation of a `try-catch` statement in the ES-CFG.

[4]. 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-CFG is shown in Fig. 3.9a.

`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 *false* arc (which let the execution flow continue to the exception exit) is non-executable (Fig. 3.9b). 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-CFG paths can be executed at runtime (Fig. 3.9c).

Example 3.5 (The ES-CFG (Part 4): *try-catch*)**.**
The program in Fig. 3.1 contains a single `try-catch`, which would be represented as shown on Fig. 3.9a (*try*) and Fig. 3.9b (*catch* that captures all exceptions thrown in `try`). In our case, the *catch* has no body, so that node will be skipped in the final ES-CFG.

Example 3.6 (The complete ES-CFG)**.**
The complete ES-CFGs of our running example is shown on Fig. 3.10. For procedure `f`, it combines the partial ES-CFGs shown on Figs. 3.4 and 3.6,

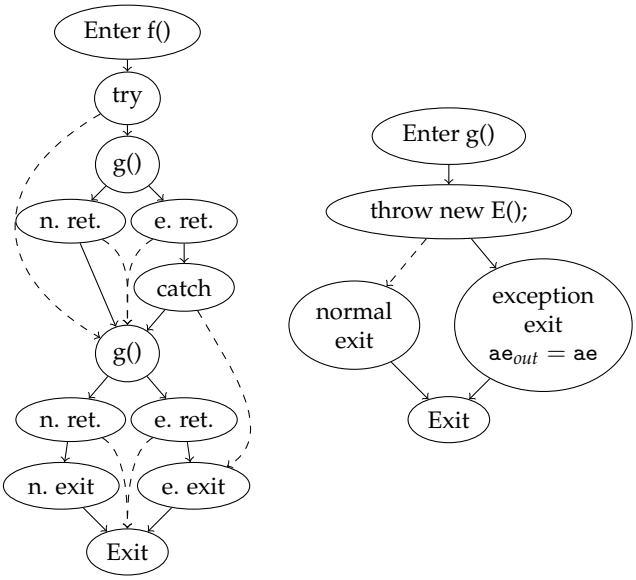


FIGURE 3.10: The ES-CFGs for the program in Fig. 3.1.

along with the `try-catch` representation from Figs. 3.9a and 3.9b. For procedure g, it uses Fig. 3.4 and the representation of an unconditional exception source (Fig. 3.7b).

3.2.2 Modifications to the APDG to create the ES-PDG

Once the ES-CFG has been generated, the next step is to generate the corresponding APDG by computing control and flow dependences. These dependences are computed in the same way as in [62] with a particular difference: while in the PDG construction shown in Section 2.4, formal-out assignments were moved to the *Enter* node after removing the *Exit* node, in the APDG, formal-out assignments remain in the corresponding *normal exit* and *exception exit* nodes (see Fig. 3.3b). The resulting graph's control dependences have slightly changed with respect to the original APDG due to the changes introduced in the ES-CFG. 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 3.1. Hence, a control dependence treatment needs to be done over the graph to classify and complement the control dependence arcs of the APDG to obtain the ES-PDG.

Algorithm 3 describes the process of adding conditional control dependence arcs to a APDG. Each dependence generates two arcs, and they are

Algorithm 3 APDG transformation

Input: $G = (N, A)$, $A_c \in A$ (control dependence)
Output: $G' = (N, A')$

1: $A_{cc1} \leftarrow \emptyset$, $A_{cc2} \leftarrow \emptyset$, $A'_c \leftarrow A_c$
2: **for all** $c \in N \mid \text{ISCATCH}(c)$ **do**
3: **for all** $(c, n) \in A_c \mid n \notin \text{STMTSINBLOCK}(c)$ **do**
4: $A'_c \leftarrow A'_c \setminus (c, n)$
5: $A_{cc1} \leftarrow A_{cc1} \cup (c, n)$
6: **for all** $n \in \text{TRYSTMTSOF}(c)$ **do**
7: **if** $\text{ISEXCEPTIONSOURCE}(n) \wedge (n, c) \in A_c^*$ **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: $A_{cc2} \leftarrow A_{cc2} \cup (c, n)$
10: $A' \leftarrow (A \setminus A_c) \cup A'_c \cup A_{cc1} \cup A_{cc2}$

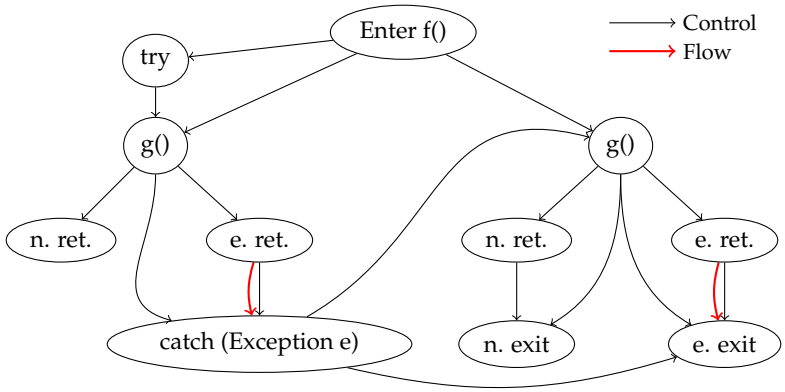
placed in two sets: CC1 and CC2. This algorithm calls methods with descriptive names. For instance, function STMTSINBLOCK, that receives a *catch* node as its argument, returns a set with all the statements in its body; and TRYSTMTSOF(c) obtains the statements in the *try* block of the given *catch*. 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.

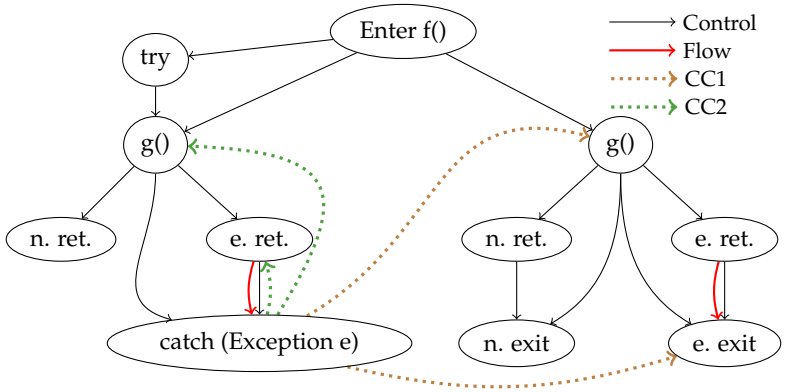
Note that conditional control dependence arcs (CC1 and CC2) are only created when the code contains at least one *catch* statement. In the case that no *catch* statement exists in the code, lines2-9 cannot be executed, and sets A'_c, A_{cc1}, and A_{cc2} reach line 10 with their initial value given in line 1, thus, $A' = A$. Therefore, the APDG and the ES-PDG are equal when there are no *catch* statements.

Example 3.7 (Computing the APDG and transformation into the ES-PDG)**.** If we use Definition 4.2 to create the APDG of procedure `f`, based on its ES-CFG, we obtain the APDG shown on Fig. 3.11a. In it, the only flow dependences correspond to the active exception, flowing from each *exception return* to the next *catch* and *exception exit*, respectively.

Then, we can apply Algorithm 3, which will analyse the only *catch* statement, and produce the following changes: the first loop selects the *catch*'s outgoing control dependences, as they point to elements outside the



(A) The APDG computed based on the ES-CFG from Fig. 3.10.



(B) The ES-PDG obtained by applying Algorithm 3 to Fig. 3.11a.

FIGURE 3.11: Application of the APDG transformation algorithm.

catch's body (which is empty), and transforms them to CC1 dependences. The second loop pairs the *catch* with all exception sources that are control dependent (directly or transitively) on the *catch* itself: the call itself and the *exception return*. Finally, it creates two CC2 dependences, which flow upwards, opposing the direction of control dependences.

3.2.3 From ES-PDGs to the final ES-SDG

The creation of the ES-SDG can be described as the union of all the ES-PDGs 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-PDG 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. 3.12, where dash-dotted arcs connect each *exit* to their corresponding *return* counterparts.

3.3 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 [54], modified later by the introduction of the pseudo-predicates and the APDG (see Algorithm 3 in [62]). 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

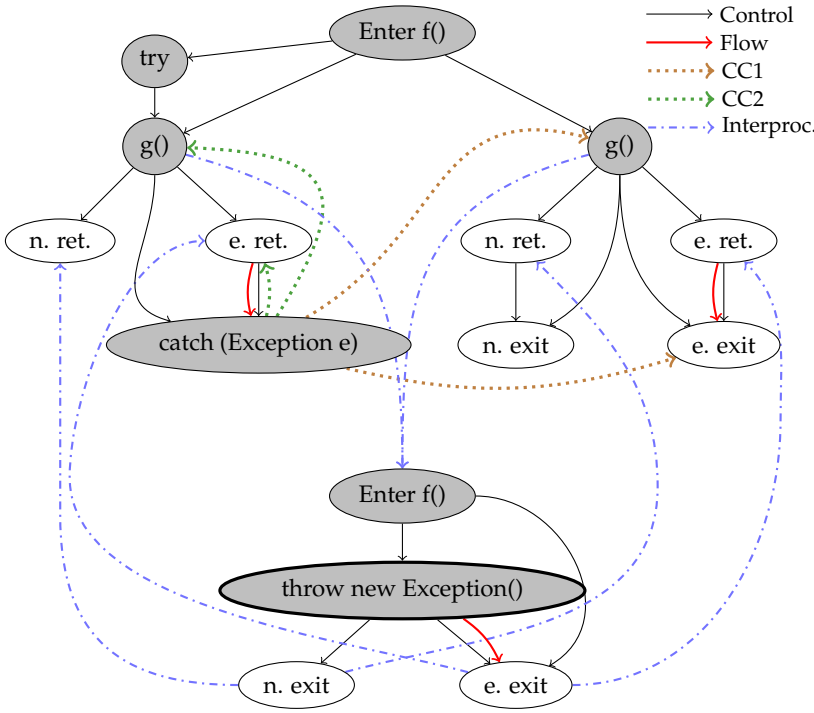


FIGURE 3.12: The ES-SDG associated to the program in Example 3.1. The slicing criterion is represented with a bold node.

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 4 illustrates how restrictions 1 and 2 are included to the SDG slicing algorithm described in Algorithm 2. 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* (line 12) or the pair $\{n, arcType\}$ does not respect the PPDG traversal restriction[2] (lines 13-14) 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 17-20). 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 24). Additionally, if the node has not been reached and added to the slice before, it is also added to the *CCExclusive* set (lines 21-23). In case the arc type is not conditional control, m is included in the slice and in the *PendingNodes* set (lines 26-29). 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 29). Finally after traversing all possible arcs, the traversal tries to include transitive dependences of CC1 arcs by a call to procedure ADDTRANSITIVECC (line 30).

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

[2]This restriction should only be included when the PPDG was the base for our graph, and not when using the APDG.

Algorithm 4 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**: $CC1R \leftarrow \emptyset, CC2R \leftarrow \emptyset, CCExclusive \leftarrow \emptyset$.

```
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 continue
            ▷ The following if should only be included when using the PPDG.
13:             if isPseudoPredicate(n) ∧ n ≠ n_sc ∧ arcType = Control then
14:                 continue
15:             m ← GETSOURCENODE(arc)
16:             if arcType = CC1 ∨ arcType = CC2 then
17:                 if arcType = CC1 then
18:                     CC1R ← CC1R ∪ m
19:                 else
20:                     CC2R ← CC2R ∪ m
21:                 if m ∈ CC1R ∧ m ∈ CC2R then
22:                     if m ∉ N then
23:                         CCExclusive ← CCExclusive ∪ m
24:                     N ← N ∪ m
25:             else
26:                 N ← N ∪ m
27:                 PendingNodes ← PendingNodes ∪ m
28:                 if m ∈ CCExclusive then
29:                     CCExclusive ← CCExclusive \ m
30:     N ← ADDTRANSITIVECC(G, N)
31:     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 continue
41:                 CC1Pending ← CC1Pending ∪ m
42:     return N
```

(line 36). If they are not in the *CCExclusive* set, the incoming *CC1* arcs are traversed iteratively adding to the slice those reached nodes that have also been reached by *CC2* arcs (lines 37-41).

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 3.8 shows the ES-SDG and the traversal of the slicing algorithm for the code of Example 3.1.

Example 3.8 (Slicing a program with *catch* correctly)**.**
If we apply Algorithm 4 to the problem shown in Example 3.1, we obtain the ES-SDG slice shown in Fig. 3.12. 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 conditional control 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.

3.4 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 called *JavaSlicer*. In order to perform a fair comparison, we implemented previous approaches

TABLE 3.1: Mean times required to generate and slice each SDG implementation.

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

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.

Chapter 9 provides details on the implementation of the slicer, how to download it, read the source and run it. *JavaSlicer* contains two approaches to exception-sensitive programs slicing: Allen and Horwitz's [4] (`AllenSDG.java`) and our own (`ESSDG.java`).

For the empirical evaluation, we strictly followed the Georges et al.'s methodology [41]. 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 3.1, 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 (*Graph creation*) show the time required to generate the graph (in ms). Finally, the last two columns (*Slicing*) 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.

3.5 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 C in some execution (a precise definition of slice can be found in Definition 2.3). The following theorem states that the slices produced by Algorithm 4 are always static backward slices, thus it is complete.

Theorem 3.1 (Completeness)**.** *Let P be a program and $G = (N, A)$ its associated ES-SDG. 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 B. 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 [4] or [55], 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 [62]. 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 arcs (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.

3.6 Related work

In Chapter 4, we explain the evolution of the SDG to treat exceptions with the definition of the ACFG and the APDG, and later the PPDG. Here, we want to complement it 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 [4], 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. [55] 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. [87], which centered around the exception system of C++, and its specific quirks and design choices; and Jie et al. [56], 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.

Chapter 4

Unconditional jumps and non-executable control flow

> I could restructure the program's flow.
> Or use one little 'goto' instead.
> Eh, screw good practice, how bad can it be?
>
> *Randall Munroe, XKCD #292*

Unconditional jumps are instructions like *goto*, *continue*, *break*, *throw*, or *return*, whose common trait is that they break the sequential execution of instructions unconditionally. To programmers, they might seem comparable to and even interchangeable with conditionals whose conditions that always produce the same value (e.g., `if (true)`). However, for static analyses (especially static program slicing) conditions are typically ignored, and every loop and branch is assumed to have two possible instructions that follow immediately after them.

The main challenge with unconditional jumps stems from the definition of control dependence (Definition 2.7): *b is* control dependent *on a if b post-dominates* ***one but not all of*** *a's* ***successors***. With this definition, only nodes with an out-degree of 2 or more (predicates) can be the source of control dependence. Unconditional jumps cannot be represented as predicates due to the fact that only one instruction could be executed after it (the jump's target) and Definition 2.5 (control-flow arcs in the CFG depend on executable sequences). Thus, unconditional jumps cannot be the source of control dependence, and without that, they cannot be included in any slice (except when they themselves are the slicing criterion).

This is not the only instance of this problem. Readers that implemented a slicer with the definitions from Section 2.4 might have noticed that the *Enter* node seems irrelevant in the PDG and SDG, as it is always disconnected from the rest of the PDG/SDG. The cause is similar to unconditional jumps: *Enter*

```
1   static int x, y;
2   static void f() {
3     if (x * 2 == y)
4       return;
5     x--;
6     System.out.println(x);
7   }
```

```
1   static int x, y;
2   static void f() {
3     if (x * 2 == y)
4       return;
5     x--;
6     System.out.println(x);
7   }
```

FIGURE 4.1: A Java program with an unconditional jump, and its PDG slice, excluding the jump.

has a single successor (the first instruction of the procedure), and has no variables for flow dependence to latch on to. It follows that *Enter* (a synthetic node) will never be included in the slice. However, in the literature, *Enter* often symbolizes the procedure header or signature, so there must be some reconciliation between the exclusion of *Enter* in all slices and it representing the procedure itself (i.e., it must be included for the procedure to be outputted in the slice).

This chapter introduces a very important concept, non-executable control-flow, which leads to a more flexible definition of control dependence. This accommodates unconditional jumps in the graph model of slicing. The data implications of unconditional jumps (the value returned or the exception thrown) will not be handled here. Return values have been covered extensively, and exceptions will be analysed in Chapter 3.

4.1 Non-executable control-flow

The CFG models the paths that the execution of a procedure may take, but fails to show what the absence of a statement represents. After all, the CFG is a general graph used in many different program analysis techniques, and slicing must contend with the effect of removing individual statements. Therefore, for slicing, we need a mechanism that models the effects of *including* an instruction and the effects of *excluding* it. The latter is rather important, as removing a jump changes the flow of the program completely. Instructions that were executed conditionally might always be executed without the jump.

Example 4.1 (Effects of removing an unconditional jump).
Consider the code in Fig. 4.1, which contains a simple loop with an unconditional jump (*return*). Line 5 (`x--`) is executed *only* when the condition on line 3 evaluates to *false*. Removing the jump changes the control-flow of the program, and thus it must be reflected in the control dependence of the

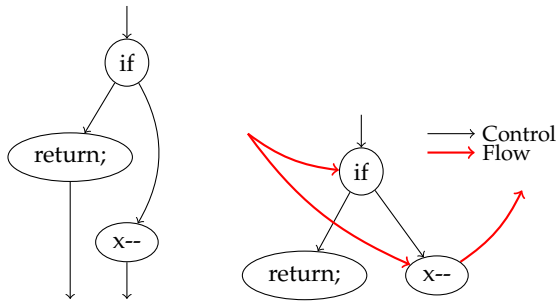


FIGURE 4.2: Partial CFG and PDG for the Java program in Fig. 4.1.

program. This is because, when producing slices, we need to make sure that instructions that are included are executed the same number of times as in the original program. Thus, we need to include unconditional jumps, unless no instructions are affected by their presence *or* absence.

To give a more specific example, consider the slicing criterion $\langle 6, \texttt{x} \rangle$, and the PDG slice shown on Fig. 4.1. The CFG and PDG for this program are shown on Fig. 4.2. The *return* statement is followed by the end of the procedure, so it only has one outgoing arc. Thus, it generates no outgoing control dependence. Its *presence* in the program creates the `if` $\rightarrow$ `x--` control dependence, but that is not enough to model the unconditional jump for slicing. Because the *absence* of a statement from the slice has not been modelled correctly, it can never be included in a traditional PDG slice (unless it is the slicing criterion).

To model the behaviour of a program when a given unconditional jump has been removed, a new kind of control-flow is defined: non-executable control-flow. In each unconditional jump, a second outgoing *non-executable* control-flow is placed, leading to the instruction that would be executed if the jump was not present.[1] An example of this pattern can be seen in Fig. 4.3c later in this chapter. The two kinds of control-flow together create *augmented* versions of the CFG and PDG.

[1]Of course, this treatment can be applied to all nodes, but it does not affect them. For example, a normal statement would have a pair of control-flow arcs connected to the next instruction. Because they target the same node, they behave (w.r.t. control dependence, Definition 2.7) as a single arc. For predicates (e.g., *if*, *while*), a similar pattern appears. Their removal would make the program skip to after its execution, but their previously present control-flow already describes this path, so it has no effect.

4.1.1 Augmented graphs (ACFG and APDG)

First, we define the augmented CFG:

Definition 4.1 (Augmented control-flow graph [8])**.** *Let P be a procedure and $G = (N, A)$ its CFG. $G' = (N, A')$ is the* augmented control-flow graph (ACFG) *of P, where $A' = A \cup A_{non-exec}$, and $A_{non-exec} \subset N \times N$ is a set of non-executable control-flow arcs such that for each $(a, b) \in A_{non-exec}$, a is an unconditional jump and b is the instruction that would be executed if a did not jump.*

With the ACFG, unconditional jumps no longer are statements, but they are not predicates either (>1 outgoing arcs in the CFG), they are called *pseudo-predicates*, to reflect the fact that they are not true predicates (one of their outgoing arcs is non-executable).

Although it is not explicitly said in Definition 4.1, the *Enter* node is considered an unconditional jump, and its non-executable control-flow arc is *Enter* $\rightarrow$ *Exit*, which will solve the problem we posed at the start of this chapter.

To create the augmented PDG we include the ACFG for control dependence, but ignore it and use the CFG for flow dependence:

Definition 4.2 (Augmented program dependence graph [8])**.** *Let P be a procedure, $G = (N, A)$ be its CFG and G'' be its ACFG. $G' = (N', A')$ is the* augmented program dependence graph (APDG) *of P, where $N' = N \setminus \{Exit\}$, $A' \subseteq N' \times N'$, and $(a, b) \in A'$ if and only if b is control dependent on a in G'' or b is flow dependent on a in G.*

The APDG uses the same slicing algorithm as the PDG (Algorithm 1), and correctly takes control dependences generated by unconditional jumps into account, producing complete slices in the presence of these jumps. For example, the APDG slice of the program shown in Fig. 4.1 w.r.t. $\langle 6, \mathtt{x} \rangle$ would be the program itself.

4.1.2 Pseudo-predicate program dependence graph

As Kumar and Horwitz noted [62], the APDG over-includes unconditional jumps, because it generates control dependences to every statement that occurs between the jump and the jump's target (either directly or transitively). Thus, they proposed a more precise definition of control dependence and a new program dependence graph.

Definition 4.3 (Control dependence in the presence of pseudo-predicates [62])**.** *Let $G = (N, A)$ be a CFG, $G' = (N, A')$ its matching ACFG, and $n_1, n_2 \in N$ be two nodes. n_2 is control dependent on n_1 if n_2 post-dominates (in G) one but not all of n_1's successors in G' ($\{n \mid (n_1, n) \in A'\}$).*

Definition 4.4 (Pseudo-predicate program dependence graph[62])**.** *Let P be a procedure, $G = (N, A)$ be its CFG. $G' = (N', A')$ is the* pseudo-predicate program dependence graph (PPDG) *of P, where $N' = N \setminus \{Exit\}$, $A' \subseteq N' \times N'$, and $(a, b) \in A'$ if and only if b is control dependent (Definition 4.3) on a or b is flow dependent (Definition 2.8) on a.*

If we compare the APDG to the PPDG, the latter contains the former, as it contains the same nodes and arcs, plus additional arcs generated by Definition 4.3. How does it produce more precise slices with more (instead of less) arcs? With an update to the slicing algorithm. Algorithm 5 is a variation of Algorithm 1, with an important restriction: if a pseudo-predicate has only been reached through a control dependence arc, the traversal does not continue.

Algorithm 5 Backward slicing algorithm with the PPDG [62]

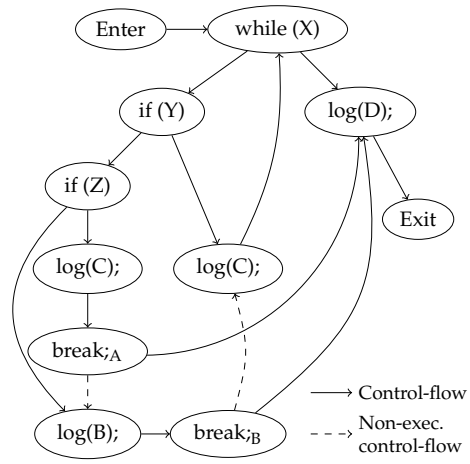
1: **function** PPDGSLICE($P, sc = \langle p, \mathcal{V} \rangle$)
2: $G = (N, A) \leftarrow$ BUILDPPDG(P)
3: $n \leftarrow$ LOCATESC(N, p)
4: $A'_c \leftarrow \{(a, b) \mid (a, b) \in A_c \wedge$ ISPSEUDOPREDICATE$(b)\}$
5: $S \leftarrow \{n' \mid (n', n) \in A\}$ ▷ Traverse a single arc from the criterion.
6: $S' \leftarrow \left\{n'' \mid (n'', n') \in (A \setminus A'_c)^* \wedge n' \in S\right\}$
7: **return** PRUNEPROGRAM(P, S')

Example 4.2 (Comparison between the APDG and the PPDG)**.**
Consider the code in Fig. 4.3a, in which a simple loop may terminate via any of the two `break` instructions. Fig. 4.3b shows its ACFG, where both `breaks` are represented as pseudo-predicates. The APDG of this code is built according to Definition 4.2, and can be seen in Fig. 4.3c. Similarly, using Definition 4.4, the PPDG of this code is the one in Fig. 4.3d. In this specific case, the APDG and PPDG match. The difference between them can be seen in the slices they produce. If we pick $\langle 11, \texttt{C} \rangle$ as the slicing criterion, the PPDG obtains a better result, as it excludes `if (Z)` and $\texttt{break}_A$. The cause of this discrepancy is the traversal limitation established by Algorithm 5, by which once the slice has reached $\texttt{break}_B$, it cannot continue, as it is a pseudo-predicate, it is not the slicing criterion, and it has only been reached by control dependences.

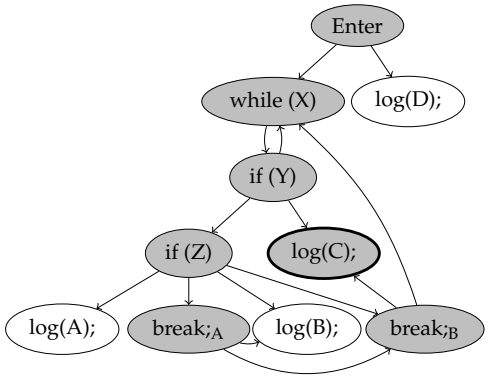
However, the PPDG is not suitable for every type of unconditional jump. Harman et al. [47] showed that it is unable to handle *goto* statements correctly in all cases, and will produce incomplete slices in some cases. It still produces complete slices with unconditional jumps used in programming languages

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

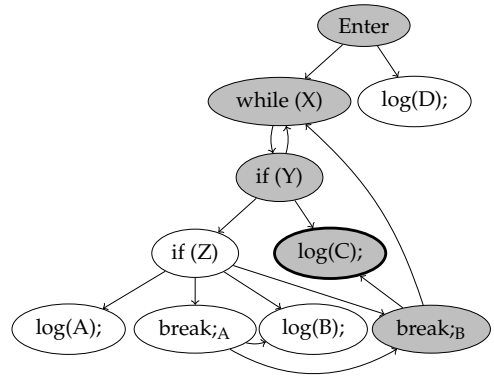
(A) A simple program with unconditional jumps.



(B) Its ACFG.



(C) Its sliced APDG.



(D) Its sliced PPDG.

FIGURE 4.3: A comparison between the APDG and the PPDG (slices are shown with grey nodes).

today (e.g., *return*, *continue*, *break*, and *throw*), so it continues to be of interest in languages or programs that do not use *goto*. According to Stack Overflow 2023 Survey [103], of the 14 programming languages used by more than 10% of professional developers, only bottom five include *goto* or equivalent instructions (C#, C++, PHP, C, and Go).

Due to the increase in precision regarding unconditional jumps, the PPDG should be considered the baseline for adaptations of the PDG. However, the balance set by Definition 4.3 and Algorithm 5 is delicate, as it assumes that:

Assumption $\mathcal{A}$. All control dependence arcs in a graph derived from the PPDG are generated according to Definition 4.3.

4.2 Interference with other slicing techniques

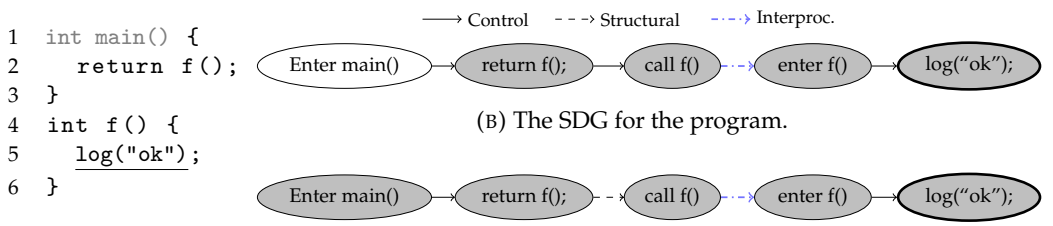
Given the ubiquity of unconditional jump instructions in imperative programming languages, every slicer targeting them should be based on either the APDG or the PPDG. improving their handling of jumps automatically. However, we have seen that the PPDG cannot be used in languages with *goto*, and in this section we show techniques that could benefit from integrating with the PPDG, but were a naive integration would result in Assumption $\mathcal{A}$ breaking down. Therefore, additional care must be taken to uphold $\mathcal{A}$, or otherwise the APDG should be used instead of the PPDG.

4.2.1 Representation of procedure calls

The sequence of graphs used in program slicing (CFG, PDG, SDG) represent most statements as a single node. Some extensions generate additional nodes to represent other elements, but the most common is the representation of calls and their arguments [53].

As described on Chapter 2, calls are represented as an additional node in the PDG and SDG, connected to the node they belong to via a control dependence arc. It is precisely the insertion of that arc that breaks $\mathcal{A}$, because it has been hand-inserted instead of generated from Definition 4.3. Example 4.3 provides a specific counter-example, where these additional control dependence arcs exclude the *Enter* node from the slice.

Example 4.3 (Erroneous program slicing of procedure calls in the PPDG). Consider the code and associated SDG shown in Figs. 4.4a and 4.4b. It contains two procedures, `main` and `f`, with one statement each. If the SDG is sliced w.r.t. the `log` statement, the traversal limitation outlined in Algorithm 5 will stop the traversal in the arc connecting "Enter `main()`" and "`return f()`" because the second one is a pseudo-predicate. Thus, code that



(A) The program.

(B) The SDG for the program.

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

FIGURE 4.4: A simple program and two variations of the SDG, sliced w.r.t. the `log` statement. The slice is represented by grey nodes.

should belong to the slice is excluded and an erroneous slice is produced.

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

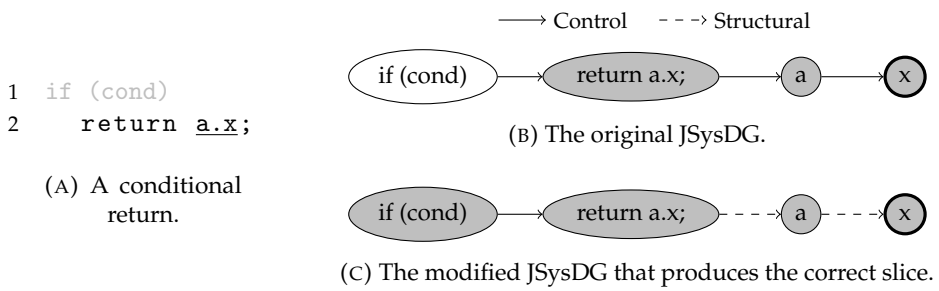
The solution would be to change the kind of arc used to connect these kinds of nodes (call nodes, argument and parameter input/output...) from control arc into any other kind. A data arc would not be semantically appropriate, so our proposal is the introduction of what we call *structural arcs*. Formally,

Definition 4.5 (Structural Arc)**.** *Given two PDG nodes $n, n' \in N$, there exists a* structural arc $n \dashrightarrow n'$ *if and only if:*

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

The first bullet point restricts the placement of structural arcs to internal structures only (and not statements that appear inside an *if*), and the second forces them to be intransitive, such that the graph forms a tree with expressions and subexpressions. Structural arcs have no restriction on their traversal, and otherwise behave as control arcs before the introduction of the PPDG's traversal restrictions.

An example of this solution has been applied to the erroneous Fig. 4.4b, producing Fig. 4.4c, a SDG where the traversal completes as expected.



(A) A conditional return.

(B) The original JSysDG.

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

FIGURE 4.5: A segment of a JSysDG and its corresponding code snippet. A slice has been produced, w.r.t. variable `a.x` in the `return` statement.

4.2.2 Object-oriented program slicing

Object-oriented programming languages are widely used, and there are program slicing techniques that adapt the SDG to include features such as polymorphism and inheritance [72]. An example is the *Java System Dependence Graph* (JSysDG), proposed by Walkinshaw et al. [105], an extension built upon the ClDG [71] (a previous attempt to include classes/objects in the SDG). Among other modifications, the JSysDG represents variables with fields as a tree of nodes. This allows the representation of polymorphism and increases its precision, allowing the selection of single fields.

Figure 4.5b showcases the structure of an object `a`, which contains a field `x`. As with procedure calls described in Section 4.2.1, the usage of control dependence arcs to represent the structure of the objects violates Assumption $\mathcal{A}$. The same solution can be applied here: using a different kind of arc to represent object structures.

Example 4.4 (Application of structural arcs to the JSysDG).
Consider again the graph in Fig. 4.5b, which has been transformed to produce Fig. 4.5c: some arcs have been turned into structural arcs (applying Definition 4.5). Now, the remaining control arc (between `if` and `return`) can be traversed by the standard algorithm and all the code is correctly included in the slice.

4.2.3 Exception handling and conditional control dependence

As we describe in Chapter 3, the introduction of exception handling necessitates the handling of unconditional jumps, specifically for the *throw* statement (or its equivalents, like *raise*). It also requires the definition of conditional control dependence (Definition 3.1) and a new slicing algorithm (Algorithm 4).

The introduction of this new dependence and its corresponding arcs break Assumption $\mathcal{A}$ in two different ways: some control dependence arcs have been specialized to become conditional control dependence (specifically, CC1), so they are no longer present in the graph; and some control dependence arcs have been inserted (CC2), so they have not been created by Definition 4.3. The consequence is that the slicing traversal of the SDG is stopped too early, and some necessary nodes are left out. This stops the traversal of control dependences too soon, leaving out of the slice the structure that contains the `try-catch`, or the *Enter* node if it is placed at the root scope of a procedure.

Example 4.5 (Conflict between exception-sensitivity and the PPDG)**.** Consider the program at Fig. 4.6, which throws and catches an exception. Its graph representation, based on Chapter 3 can be seen at Fig. 4.6b. If we produce a slice, using Algorithm 4, starting at $\langle 4, \emptyset \rangle$, we would immediately reach "try" and "throw". Then, because we reach "catch" simultaneously from a CC1 and CC2 dependence, we also include it. The traversal stops there, as all remaining control dependences cannot be traversed: *Enter* $\rightarrow$ "try" and *Enter* $\rightarrow$ "throw" because the target is a pseudo-predicate that has only been reached via control dependence, and *Enter* $\rightarrow$ "catch" because the target has been reached via conditional control dependence (recall that Algorithm 4 stops the traversal when a node is reached via conditional control dependence). The resulting (incorrect) slice can be seen in the grey-out nodes in Fig. 4.6b.

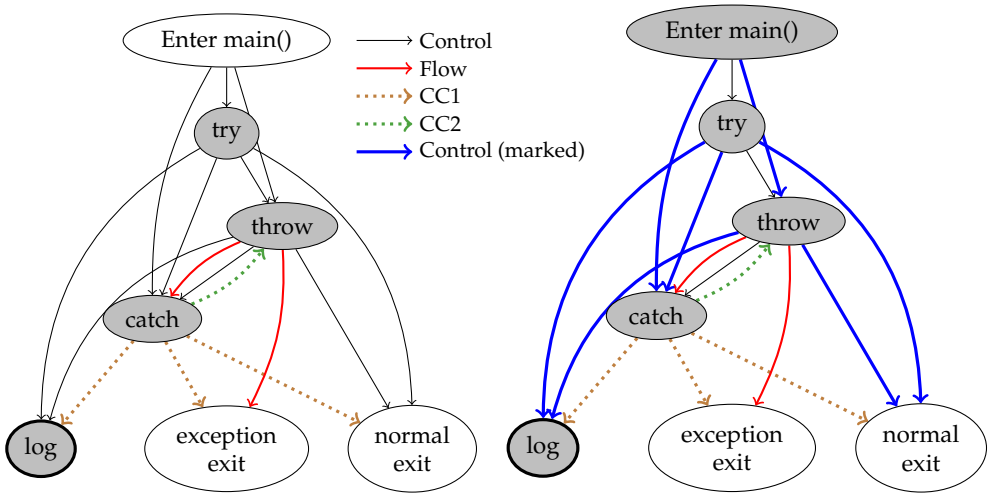
Solving this issue is much more complicated than the previous two (calls and object representation), as the source of the problem is quite different. In this case, some control dependences have effectively been removed from the graph. Thus, the solution is to partially relax the traversal restrictions that the PPDG imposes on control dependence arcs.

First, the difference between the PPDG and the PDG must be computed. They share the same set of nodes, so this difference can be expressed as a set difference between the two sets of arcs. Arcs that are only present in the PPDG and not in the PDG must be marked. Arcs that are marked are not considered control dependences for the purposes of traversal restrictions from the PPDG. This change solves the problem, and the traversal continues normally, reaching the *Enter* node.

Example 4.6 (Fixing slicing with exceptions and unconditional jumps)**.** Consider the amended graph at Fig. 4.6c, which is obtained by marking the control arcs in Fig. 4.6b that are exclusive to the PPDG in bold blue. Then, when a slice is computed starting at "log", "try" would be reached via a marked arc, and the traversal would continue back to the *Enter* node,

```
1 void main() {
2   try { throw e1; }
3   catch (T e2) {}
4   log(42);
5 }
```

(A) A program that throws and catches an exception.



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

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

FIGURE 4.6: A simple program that throws an exception and two SDGs with conditional control dependence (CCD), sliced w.r.t. $\langle 4, \emptyset \rangle$.

producing a correct slice.

In general, this solution works because it relaxes the traversal restrictions of the PPDG slightly. Recall that the restriction forbids traversing control dependences whose target is a pseudo-predicate, but the ES-SDG converts some of the computed control dependences into conditional control dependences (specifically, CC2). If this restriction is relaxed completely, the PPDG would behave like the APDG, losing the precision gained. However, relaxing the condition on control arcs that are present only on the PPDG provides a reasonable compromise between the precision improvement of the PPDG and fixing the incompleteness introduced by the ES-SDG (when it breaks $\mathcal{A}$).

4.3 Implementation in JavaSlicer

The incompatibility between different slicing models was detected during the implementation of a slicer that included all the described models. To solve the detected problems, the solutions proposed throughout this chapter have been implemented as the base of our JavaSlicer, which is described in more detail on Chapter 9. It contains and implements an SDG that covers parts of the Java programming language.

Solving most problems presented in Section 4.2 only requires changing the kind of arc used (represented with the `StructuralArc` class). However, for exception handling (Section 4.2.3), two changes are required. The first is computing the arcs that are present in the PPDG but not in the PDG, which is done in the `PPDG` class:

PPDG.java

```
47 /** Finds the CD arcs that are only present
48  * in the PPDG and marks them as such. */
49 protected void markPPDGExclusiveEdges(CallableDeclaration<?> declaration) {
50   APDG apdg = new APDG();
51   apdg.build(declaration);
52   Set<Arc> apdgArcs = new HashSet<>();
53   for (Arc arc : apgg.edgeSet())
54     if (arc.isUnconditionalControlDependencyArc())
55       apdgArcs.add(arc);
56   for (Arc arc : edgeSet())
57     if (arc.isUnconditionalControlDependencyArc() && !apdgArcs.contains(arc))
58       arc.asControlDependencyArc().setPPDGExclusive();
59 }
```

This method is called during the creation of the PPDG, upon which the exception-sensitive graphs (ES-PDG and ES-SDG) are based. Then, in the

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

ExceptionSensitiveSlicingAlgorithm.java

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

4.4 Related work

There exist very few works that reason about the consequences of integrating different program slicing techniques in the same model. Unfortunately, the solutions proposed for the different slicing problems were mostly proposed in isolation of other problems and solutions. This means that every time a new challenging slicing problem is solved, the solution is presented as an evolution of a previous program representation used to solve the same problem (if there is a previous one) or from the original PDG/SDG representations. For example, this is the case of OO languages where the graph proposed by Larsen and Harrold in [71] was further extended by Liang and Harrold in [72], and lately used by Walkinsaw et al. in [105] to represent Java object-oriented programs. But none of them considered, e.g., exception handling; as they assumed that the standard solution to exception handling would not interfere with their solution to OO. The real world is however a continuous interference between problems and solutions. Any implementation for a real programming language must include different solutions and they will necessarily interfere.

Most solutions proposed in the literature are based on the original SDG and the original slicing algorithm proposed in [53]. Many proposals enhance the expressivity of the graph in different ways: adding modifications to the CFG used to generate the final graph ([30, 62]) or including new program dependences ([14, 15, 27, 111], and sometimes these new models come with some modifications to the slicing traversal algorithm to improve computed slices ([27, 30, 61, 62]).

Since the initial graph of all these proposals is the original SDG, most of the solutions designed to slice other previously solved problems are not considered when designing a new program representation model. This is the case of several object-oriented program representations like [71, 72], where the

SDG is augmented to represent polymorphic method calls with the addition of new graph nodes that consider the statically undecidable method called. Another example is the one presented by Cheng [14, 15, 111] for program slicing of concurrent programs. Cheng's proposal starts from the PDG and defines a set of specialised dependences required in concurrent environments, building a representation called the *program dependence net* (PDN).

Taking a look at the literature, we consider whether a solution to a particular slicing problem can be classified as control dependence conservative (i.e., when all control arcs in the graph are computed over Definitions 2.7 and 4.3) or not. For example, the works presented by Krinke and Snelting [60], Cheda et al. [13], and the one presented by Kinloch and Munro [57] are all control dependence conservative. While Krinke and Snelting, and Cheda et al. define a new type of arc to represent what we call structural dependences, Kinloch and Munro construct the PDG by splitting assignments in different nodes (left-hand side and right-hand side) controlled by the same node. On the other hand, other approaches are not control dependence conservative, like the ones mentioned for object-oriented programs ([71, 72, 105]) because they arbitrarily represent the unfolding of object variables with the use of control arcs not computed by Definitions 2.7 and 4.3.

The concept of structural dependence was already noted by previous authors along the literature. Krinke and Snelting [61], in their fine-grained program representation, decomposed program statements and noted a new kind of dependence they called *immediate control dependence* which, in fact, is similar to our defined structural dependence. Unfortunately, the representation of Krinke and Snelting cannot be applied to the SDG because of its granularity level. Their fine-grained PDG split every single statement in a set of nodes and this immediate control dependence was only usable for connecting the different parts of the statement. Additionally, the main purpose of the immediate control dependence during the slicing phase was to avoid the appearance of loops by limiting the slicing traversal. Also Cheda et al. [13], in their approach to static slicing for first-order functional programs which are represented by means of rewrite systems, designed the *term dependence graph* (TDG), that includes a set of arcs labelled with *S* for being considered *Structural* when representing the dependence between method calls and their corresponding arguments. Their TDG is only usable for first-order functional programs which are represented by means of rewrite systems and is not a derivation of the PDG. The idea of considering structural arcs to connect calls and arguments is promising if applied to SDG program slicing, but it only solves one of the incompatibilities detected.

Chapter 5

Communicating Sequential Processes and execution modelling

> Wait a minute, Doc. Ah... Are you telling me you built a time machine... out of a DeLorean?
>
> *Back to the Future (1985)*

One of the most extended languages to model concurrent processes is *Communicating Sequential Processes* (CSP) [51, 97]. The specification, analysis, and transformation of CSP specifications have often been based on the use of *CSP traces* [97], which represent the sequences of events that can occur in a system. In fact, there are different analyses such as deadlock analysis [63, 108], livelock analysis [16], and security analysis [23], among others, that are based on or use this kind of traces.

Example 5.1 (Ambiguous CSP traces).
Consider the following CSP specification (an introduction to CSP is available on Section 5.1):

```
channel a,b
MAIN = P ||{a} Q
P = a → b → SKIP
Q = b → a → ((b → SKIP) □ Q)
```

In this specification, two kinds of event can be emitted, one per channel. The main process starts, and runs `P` and `Q` in parallel, synchronizing on event `a`. Process `P` simply emits `a`, then `b`, and finishes. However, `Q` is more complicated, including recursion. First, it emits `b`, then `a`. Afterwards, it

performs a non-deterministic or external choice between emitting `b` and finishing, and recursively calling itself.

The only possible traces of this specification are $\langle\rangle$, $\langle$`b`$\rangle$, $\langle$`ba`$\rangle$, $\langle$`bab`$\rangle$, $\langle$`babb`$\rangle$. If we consider the last trace, it can be produced by four different computations due to the non-deterministic evaluation order of the processes. While the first two events (`ba`) are deterministic (i.e., the `b` event corresponds to the first `b` in process `Q`, and the `a` event corresponds to the synchronization of the `a` events in `P` and `Q`), the last two `b` events in the trace can be produced by `P` and one of the two branches of `Q` interleavedly.

Unfortunately, standard CSP traces are not adequate for analyses that need to link the trace with the source code (e.g., debugging). The trace of a computation does not always provide enough information to identify a bug in the source code, or even to decide whether the computation is buggy. For instance, if we execute the program in Example 5.1 and we get the trace $\langle$`babb`$\rangle$, we cannot know what parts of the code have been executed. It could be the case that process `Q` is buggy (e.g., it should be `Q = b` $\rightarrow$ `a` $\rightarrow$ `(c` $\rightarrow$ `SKIP` $\Box$ `Q)`). But, even if the buggy code of `Q` (i.e., `b` instead of `c`) was actually executed, we would not know it, because the trace produced by the buggy code is identical to a correct trace (following another path). The problem still remains if the generated trace is wrong. For instance, if the last event of the trace (`b`) is buggy we cannot know what part of the program produced this buggy event (in fact, it could be any of the three `b` events in the specification). And, what is even more important, if we are interested in a particular buggy event, we cannot trace the error backward (reversing the computation [1]). A proper reversibility tool for CSP would allow programmers to undo unproductive computations, and to backtrack to a save state when an error is detected.

In this chapter, we propose an operational semantics of CSP that is (i) conservative (i.e., the forward evaluation follows the standard operational semantics of CSP), and that (ii) produces as a side effect the *history of the computation* such that every evaluated expression has information associated with the source code. In this way, (1) we can know exactly what expressions of the program are evaluated at each step (i.e., we can associate the events that occur in the computation with the corresponding literals in the source code), and (2) the history of the computation allows us to reverse the computation. The proposed model has been implemented in a system to animate CSP computations forward and backward.

Domains

$M, N \ldots \in \mathcal{N}$ (Process names) $\quad P, Q \ldots \in \mathcal{P}$ (Processes)

$a, b \ldots \in \Sigma$ (Events) $\quad u, v \ldots \in \mathcal{V}$ (Variables) $\quad \overline{EV_n} = EV_1, \ldots, EV_n$

S	::=	$\{D_1, \ldots, D_n\}$	(Entire specification)
D	::=	$M = P$	(Process definition)
	|	$M(\overline{EV_n}) = P$	(Parameterized process)
P	::=	M	(Process call)
	|	$M(\overline{EV_n})$	(Parameterized process call)
	|	$CO \rightarrow P$	(Prefixing)
	|	$P \sqcap Q$	(Internal choice)
	|	$P \,\square\, Q$	(External choice)
	|	$P \not< Bool \not> Q$	(Conditional choice)
	|	$P \,;\, Q$	(Sequential composition)
	|	$P \underset{\{\overline{EV_n}\}}{\|\|} Q$	(Synchronized parallelism)
	|	$SKIP$	(Skip)
	|	$STOP$	(Stop)
CO	::=	$EV \mid CO?EVI \mid CO!EV$	(Compound Object)
EVI	::=	$EV \mid v : b$	(Input event with Variables)
EV	::=	$a \mid v \mid EV.EV$	(Event with Variables)
$Bool$	::=	$true \mid false \mid Bool \wedge Bool$	(Boolean expression)
	|	$Bool \wedge Bool \mid \neg Bool$	
	|	$EV = EV \mid EV \neq EV$	

FIGURE 5.1: Syntax of CSP specifications

5.1 A brief introduction

In this section we introduce some notation, and provide definitions used in the rest of the chapter. Fig. 5.1 summarizes the CSP syntax constructs that we consider in this chapter. Extended explanations for each syntax construct can be found in [97]. A CSP *specification* is viewed as a finite set of process definitions. The left-hand side of each definition is the name of a process, which is defined in the right-hand side by means of an expression formed from the operators in Fig. 5.1 (see Examples 5.1 and 5.8). The operational semantics of CSP is an event-based semantics, where the occurrence of events fire the rules (readers non-familiar with the standard operational semantics of CSP can find a detailed description in [97]). We use the domains Σ and Π that, respectively, contain all external (i.e., visible from the external environment) and internal events in the execution of a CSP specification.

To uniquely identify each literal in a CSP specification we use labels (that we call *program positions*). Each program position corresponds to a unique

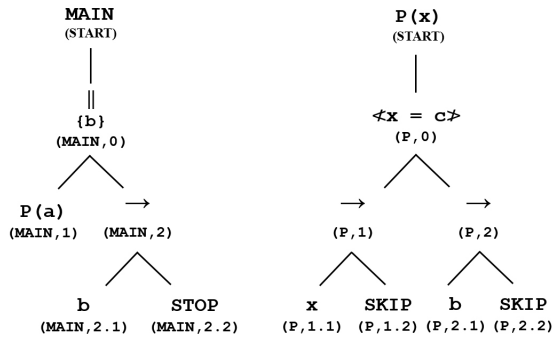


FIGURE 5.2: Program positions of the program in Example 5.2.

node in the CSP specification's abstract syntax tree (AST). A *program position* [75] is a pair `(P,s)` where `P` is the name of a CSP process and `s` is a sequence of natural numbers. The root of the AST is represented with `0`, and, for each operator, the operands are numbered from left to right. We use a special label `(START)` to represent the initial call to a process in the program. We define the domain $\mathcal{P}os$ to represent the set of all possible program positions. We also define $\mathcal{P}os_\Sigma \subset \mathcal{P}os$ as the subset of positions that refer to external events (i.e., events in Σ).

Example 5.2 (Labelling a CSP program).
The following CSP program has been labelled with program positions (they are underlined):

$$\texttt{MAIN} = \texttt{P(a)}_{\underline{(\texttt{MAIN},1)}} \underset{\{\texttt{b}\}}{\parallel}_{\underline{(\texttt{MAIN},0)}} (\texttt{b}_{\underline{(\texttt{MAIN},2.1)}} \rightarrow_{\underline{(\texttt{MAIN},2)}} \texttt{STOP}_{\underline{(\texttt{MAIN},2.2)}})$$
$$\texttt{P(x)} = (\texttt{x}_{\underline{(\texttt{P},1.1)}} \rightarrow_{\underline{(\texttt{P},1)}} \texttt{SKIP}_{\underline{(\texttt{P},1.2)}}) \not\leftarrow \texttt{x} = \texttt{c} \not\rightarrow_{\underline{(\texttt{P},0)}} (\texttt{b}_{\underline{(\texttt{P},2.1)}} \rightarrow_{\underline{(\texttt{P},2)}} \texttt{SKIP}_{\underline{(\texttt{P},2.2)}})$$

All terms are uniquely labelled because labels help maintain the order of the associated AST (see Fig. 5.2).

5.2 Recording the history of a CSP computation

Following the Landauer's embedding principle [64], one can make a CSP computation reversible by recording the history of the computation. In this section, we discuss the information that must be stored to record a CSP computation and make it reversible.

We propose an extension of CSP tracks [75]—a data structure used to represent (forward) CSP computations—to store the execution history of CSP

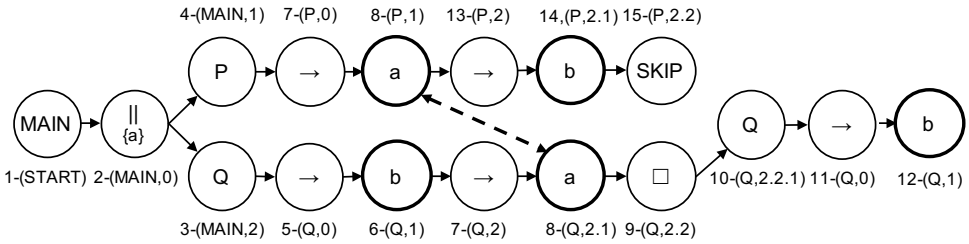


FIGURE 5.3: R-track of a computation of the program in Example 5.1.

computations, and based on it, we present a formulation and implementation of a system to reverse CSP computations.

A CSP track is a dynamic data structure that represents the sequence of expressions that have been evaluated during one computation, labelled with the location of these expressions in the specification. In contrast, a (standard) CSP trace is the sequence of events that occur during the computation [97]. Therefore, a CSP track is much more informative than a CSP trace because the former not only contains a lot of information about original program structures but it also explicitly relates the sequence of events with the parts of the specification that caused these events. Our reversibility extension of a track is called an R-track. R-tracks extend tracks with two small changes: (i) nodes are labelled with timestamps that represent the time when the expression associated with each node was produced, and (ii) the representation of prefixing is changed (in a track, `b -> SKIP` is represented with three sequential nodes: `(b)` $\rightarrow$ `(->)` $\rightarrow$ `(SKIP)`; while in an R-track it is represented as `(->)` $\rightarrow$ `(b)` $\rightarrow$ `(SKIP)`, i.e., the order of the prefix and the prefixing operator is reversed so that we can undo the prefixing operator when the event is undone).

Example 5.3 (R-tracks)**.**
The data structure in Fig. 5.3 is the R-track associated with a computation of the program in Example 5.1. This computation selects the recursive branch in the external choice of process `Q`. Moreover, it executes the recursive call to `Q` before executing the `b` event in process `P`. Hence, the trace produced is $\langle$`babb`$\rangle$.

In Fig. 5.3 we can also see that every single literal of the program evaluated in a computation is recorded inside a node of the associated R-track. Moreover, each node has a label with two components `t-p`, where `t` is a timestamp that represents the instant where this node was generated, and `p` is the *program position* of the literal in this node. For the sake of clarity, when it is clear from

the context, we often use the internal literal instead of the program position (i.e., we use `(8-a)` instead of `(8-(P,1))`).

With the timestamp we can serialize the program. For instance, if we only focus on event nodes (those with a bold line) then it is trivial to generate the associated trace $\langle$`babb`$\rangle$ following the sequence: `(6,b)`$\rightarrow$`(8,a)`$\rightarrow$`(12,b)`$\rightarrow$`(14,b)`. Observe also that the R-track records explicit information about synchronizations, which are represented with a dashed arc between the two synchronized events. Synchronizations among three or more events are represented with pairwise arcs. It is also important to remark that R-tracks not only record events in Σ but also internal events (i.e., $\Sigma \cup \Pi$). However, for the sake of simplicity and readability, we only consider events in Σ, because the extension to Π is straightforward (but verbose).

Once we have available a data structure such as the R-track, three research questions emerge: (1) Can R-tracks be used to reverse a CSP computation? Yes. We discuss how to use R-tracks to reverse computations in Section 5.2.1. (2) Can R-tracks be automatically generated from CSP computations? Yes. We have implemented a conservative extension of the standard semantics that can generate tracks efficiently from CSP computations. This is described in Section 5.3. (3) Is the use of R-tracks scalable to real programs? Yes. We discuss scalability issues in Section 5.3.1.

5.2.1 Reversing CSP computations with R-tracks

An R-track contains enough information to (re)execute the program both forward and backward in the standard way (i.e., producing a derivation of rewriting steps following the rules of the operational semantics). However, we want to go beyond the standard execution. We want to directly map the execution to the source code. So that, every single event fired along the execution is associated with a set of expressions of the source code. With this view, a derivation of a program is a sequence of pairs `(e,p)` where `e` is an event, and `p` is the set of program positions that point to the expressions in the source code needed to fire this event. We call such a derivation an *event–syntax trace*.

Example 5.4 (Tracing an output with R-tracks).
Consider again the CSP program in Example 5.1. One possible event–syntax trace of this program is:

```
(b, {(START), (MAIN,0), (MAIN,2), (Q,0), (Q,1)})
→ (a, {(MAIN,1), (P,0), (P,1), (Q,2), (Q,2.1)})
→ (b, {(Q,2.2), (Q,2.2.1), (Q,0), (Q,1)})
→ (b, {(P,2), (P,2.1)})
```

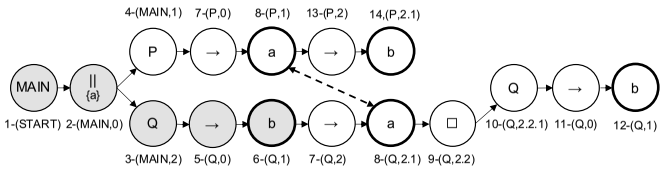
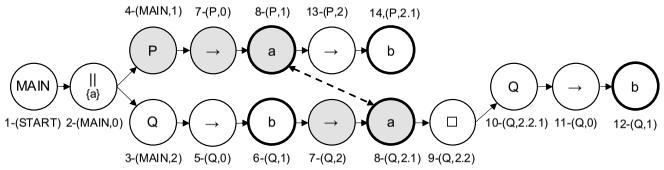
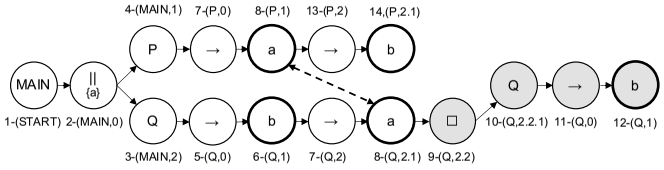
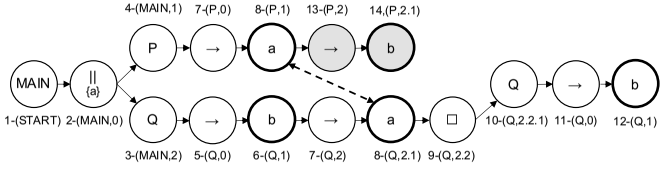
Event	Running R-track (executed part in grey)	Event–syntax rewriting step
b		$(\mathcal{G},0) \xrightarrow{\{(START),(MAIN,0),(MAIN,2),(Q,0),(Q,1)\}} (\mathcal{G},6)$
a		$(\mathcal{G},6) \xrightarrow{\{(MAIN,1),(P,0),(P,1),(Q,2),(Q,2.1)\}} (\mathcal{G},8)$
b		$(\mathcal{G},8) \xrightarrow{\{(Q,2.2),(Q,2.2.1),(Q,0),(Q,1)\}} (\mathcal{G},12)$
b		$(\mathcal{G},12) \xrightarrow{\{(P,2),(P,2.1)\}} (\mathcal{G},14)$

FIGURE 5.4: Event-syntax trace associated with the trace $\langle$babb$\rangle$.

This event-syntax trace is graphically represented in the second column (Running R-track) of Fig. 5.4. There we can see that each set of grey nodes represent the portion of code that is needed to fire the associated event (represented with a bold line). With this information, we can highlight the source code that is being executed in each step, and we can do it both forward and backward.

Event–syntax traces are particularly useful for debugging, because they explicitly show what exact expressions in the source code participate in each step of the computation. Hence, one could execute the program (either forward or backward) and highlight in the source code all parts that participate in the firing of an event. To formalize the generation of event–syntax traces, we can define an R-track rewriting semantics that iteratively transforms the information of an R-track with every event of the computation. We need to provide first some formal definitions.

Definition 5.1 (R-track)**.** *An R-track is a labelled directed acyclic graph* $\mathcal{G} = (N, A_c, A_s)$ *where* N *are the nodes, and arcs are divided into two groups:*

- *control-flow arcs* (A_c) *are a set of one-way arcs (denoted with* $\mapsto$*) representing the control-flow between two nodes, and*
- *synchronization arcs* (A_s) *are a set of two-way arcs (denoted with* $\leftrightarrow$*) representing the synchronization of two (event) nodes.*

Each node $n \in N$ *has two labels:* $time(n)$ *contains a natural number (the timestamp), and* $pos(n)$ *contains a program position. Given two nodes* $n, n' \in N$*, the CSP term* $pos(n)$ *is executed before* $pos(n')$ *if and only if* $time(n) < time(n')$*.*

For the sake of clarity, we have included inside the nodes of Figs. 5.3 and 5.4 a label with a source code literal. Note, however, that this label does not exist in the definition of R-track because it is actually redundant with respect to the label *pos*, since the program position uniquely identifies the literal. To ease the reading, we use a node n to denote the program expression that n actually represents. Hence, we use, e.g., $n \in \Sigma$ instead of $pos(n) \in \mathcal{P}os_\Sigma$, so that the notation becomes more readable. This notion of R-track is a slight conservative extension of the standard tracks. How to compute tracks from CSP specifications is described in [76], where the correctness of tracks is also proved in Theorem 3.

Because we want to execute R-tracks, we need a mechanism to know what part of the R-track has already been executed. This idea is captured with the notion of *Running R-track*.

Definition 5.2 (Running R-track)**.** *A* running R-track *is a pair* $(\mathcal{G}, t)$*, where* $\mathcal{G} = (N, A_c, A_s)$ *is an R-track and* t *is a natural number (a timestamp) such that* $t = 0$ *or* $\exists n \in N \,.\, time(n) = t \wedge n \in \Sigma$*.*

The timestamp represents the last executed event. If the timestamp is 0, then the execution has not started yet. For instance, in the track of Fig. 5.3, the timestamp 8 represents the occurrence of event `a`.

Given a running R-track $(\mathcal{G}, t)$ with $\mathcal{G} = (N, A_c, A_s)$, we define functions *first-after*$(\mathcal{G}, t)$ and *last-before*$(\mathcal{G}, t)$ to obtain the first (respectively last) event node of the R-track after (respectively before) the given timestamp t. Formally,

$$\begin{aligned} \textit{first-after}(\mathcal{G}, t) = \{n \mid\ & t < time(n) \\ & \wedge \nexists n' \,.\, t < time(n') < time(n) \\ & \wedge n, n' \in \Sigma \wedge n, n' \in N\} \\ \textit{last-before}(\mathcal{G}, t) = \{n \mid\ & time(n) < t \\ & \wedge \nexists n' \,.\, time(n) < time(n') < t \\ & \wedge n, n' \in \Sigma \wedge n, n' \in N\} \end{aligned}$$

Note that both *first-after* and *last-before* return a set of nodes because, due to synchronizations, there could be different nodes that are the first (respectively last) nodes to be executed (all of them at the same time). A clear example of this phenomenon happens in Fig. 5.3. The last event nodes before timestamp 14 are `8-(P,1)` and `8-(Q,2.1)` (the synchronization of `a`).

We also need to define a causality relation between nodes.

Definition 5.3 (Causality Relation in R-tracks)**.** *Given an R-track* $\mathcal{G} = (N, A_c, A_s)$ *and two nodes* $n_1, n_2 \in N$*, we say that* n_1 *is causal with respect to* n_2 *(denoted* $n_1 \rightsquigarrow n_2$*) if and only if* $(n_1, n_2) \in A^*$*, where* A^* *denotes the reflexive and transitive closure of* A_c *and* A_s *(*$A^* = (A_c \cup A_s)^*$*).*

The causality relation is transitive and it is induced by control-dependence and synchronizations. It is useful to determine what nodes in the R-track participate in each rewriting step. For this, we can divide the R-track into complementary sections containing a single event occurred in time, and all causal nodes of this event that are not causal with respect to other previous events (see Fig. 5.4). This is done with function *causalNodes*(n). Roughly, it returns those nodes that are causal with respect to n (i.e., they precede n in a control path) and they do not precede an event node that happened before n in the same control path. Formally, given a running R-track $(\mathcal{G}, t)$ with $\mathcal{G} = (N, A_c, A_s)$, and a node $n \in N$, we define *causalNodes*(n) as:

$$causalNodes(\mathcal{G}, n) = SYNC_n \cup \{n' \in N \mid n' \notin \Sigma \\ \wedge (n' \mapsto n_s) \in A_c^* \\ \wedge (\nexists n'' \in \Sigma \,.\, n'' \notin SYNC_n \\ \wedge (n' \mapsto n''), (n'' \mapsto n_s) \in A_c^*)\}$$

where $n_s \in SYNC_n = \{n\} \cup \{n_{sync} \mid (n \leftrightarrow n_{sync}) \in A_s\}$, and we use A_c^* to denote the reflexive and transitive closure of A_c.

Example 5.5 (Causal node relationships)**.**
Fig. 5.4 displays four R-tracks. In the first, third, and fourth R-tracks, all grey nodes are the causal nodes of the last grey nodes. In the second R-track, however, not all causal nodes of the last grey node are in grey. The causal nodes of the nodes with timestamp 8 are all grey nodes plus nodes with timestamps 1 and 2.

Two nodes can share the same causal nodes. For instance, in Fig. 5.3, nodes 1 and 2 are causal with respect to nodes 3 and 4. Given the occurrence of an event in an execution, we want to identify those nodes that caused this event *since the occurrence of the last event*. Therefore, we also need to define a function $stepNodes(n)$ that captures this restricted causal relation in the R-track. Roughly, it returns those nodes that precede n in the R-track and do not precede an event node that happened before n.

$$stepNodes(\mathcal{G}, n) = causalNodes(\mathcal{G}, n) \setminus \\ \{n' \in N \mid (n' \mapsto n'') \in A_c^* \wedge n'' \in \Sigma \wedge time(n'') < time(n)\}$$

Example 5.6 (Step nodes in R-tracks)**.**
In Fig. 5.4, four R-tracks are shown. In each R-track, all grey nodes are the step nodes of the last grey nodes.

We are now in a position to define a calculus able to reproduce a computation both forward and backward from an R-track (i.e., without the need of the source code program, and without the need of the standard semantics).

Given an R-track $\mathcal{G} = (N, A_c, A_s)$, the application of the following *forward rule* is able to produce an event–syntax trace reproducing the computation forward:

(*Forward*)

$$\frac{n \in \mathit{first\text{-}after}(\mathcal{G}, t) \quad t < t' = \mathit{time}(n) \quad P = \{\mathit{pos}(n') \mid n' \in \mathit{stepNodes}(n)\}}{(\mathcal{G}, t) \xrightarrow{P} (\mathcal{G}, t')}$$

Analogously, the reverse counterpart to produce a backward derivation can be done with the application of the *backward* rule:

(*Backward*)

$$\frac{\mathit{time}(n) = t > t' \quad n' \in \mathit{last\text{-}before}(\mathcal{G}, t) \quad t' = \mathit{time}(n') \quad P = \{\mathit{pos}(n'') \mid n'' \in \mathit{stepNodes}(n)\}}{(\mathcal{G}, t) \xleftarrow{P} (\mathcal{G}, t')}$$

It is important to remark that the direction of the steps of the semantics always go from left to right. The direction of the arrow only indicates whether the step is forward or backward.

Example 5.7 (Applying forward and backward rules)**.**
Let $\mathcal{G}$ be the R-track in Fig. 5.3. The application of four rewriting steps with the *forward rule* would produce the derivation shown in the right column of Fig. 5.4. The associated reverse computation is completely symmetric.

Deterministic reversibility of CSP

The technique proposed so far is a mechanism to deterministically execute forward and backward a computation strictly following the order of the original execution. That is, at any state, there is at most one possible forward execution step, and at most one possible backward execution step. Formally,

Theorem 5.1 (Deterministic forward and backward execution)**.** *Given a running R-track* $(\mathcal{G}, t)$*,*

- $(\mathcal{G}, t) \xrightarrow{P} (\mathcal{G}, t') \wedge (\mathcal{G}, t) \xrightarrow{Q} (\mathcal{G}, t'') \implies P = Q \wedge t' = t''$
- $(\mathcal{G}, t) \xleftarrow{P} (\mathcal{G}, t') \wedge (\mathcal{G}, t) \xleftarrow{Q} (\mathcal{G}, t'') \implies P = Q \wedge t' = t''$

Proof. Trivial by definition, since t' (and thus t'') is unique because functions *first-after*, *time*, *pos*, and *stepNodes* are deterministic functions; and hence (*Forward*) and (*Backward*) are also deterministic. □

Moreover, at any state s_1 where the execution of a forward step produces a new state s_2, then the execution of a backward step at s_2 always produces the original state s_1. Formally,

Theorem 5.2 (Symmetric reversibility)**.** *Given a running R-track* $(\mathcal{G}, t)$*,*

- $(\mathcal{G}, t) \xrightarrow{P} (\mathcal{G}, t') \xleftarrow{Q} (\mathcal{G}, t'') \implies P = Q \wedge t'' = t$

Proof. Since *first-after*$(\mathcal{G}, t) =$ *last-before*$(\mathcal{G}, t')$ for $n \in$ *first-after*$(\mathcal{G}, t)$ and $t' =$ *time*(n) by definition, the claim holds. □

Causal-consistent reversibility of CSP

Causal-consistent reversibility [67] was first introduced in RCCS [18], a reversible variant of CCS. This form of reversibility empowers the idea that reversibility may not be necessarily symmetric in a concurrent environment provided that causality is kept consistent after every reversible step (hence, the name causal-consistent).

The main idea is that independent parallel processes should be reversed independently (thus keeping the essence of concurrence, where independent events can be executed in any order). Contrarily, when causal dependencies exist between parallel processes (e.g., synchronizations) they must be taken into account when reversing these processes. In particular, any action can be undone provided that all its consequences, if any, are undone beforehand. For instance, if two processes synchronize on event `a`, then the events that happened before `a` in any of the processes cannot be undone until all the events that happened after `a` in both processes have been undone.

Clearly, our reverse execution rule (*Backward*) is too restrictive for causal-consistent reversibility. Even though all backward steps made by this rule are always causal-consistent, there are many possible backward schedules that are also causal-consistent but they are not allowed by the rule. For instance, in Example 5.7, the last two `b` events could be causal-consistently reversed in any order because they belong to processes that run in parallel and they are independent after the synchronization of event `a`.

In order to define proper rules for both causal-consistent forward and backward execution, we first need to define a causal-consistent version of functions *first-after* and *last-before*. Because a causal-consistent step does not necessarily follow the execution order of the R-track, the timestamp cannot be used anymore to know in what point of the execution we are (for instance, in Fig. 5.3 timestamp 14 could be causal-consistently replayed before timestamp 12, thus changing the original order of events). Therefore, instead of timestamps we use a set with the already executed nodes N^{ex} of an R-track $\mathcal{G} = (N, A_c, A_s)$ (i.e., $N^{ex} \subseteq N$). Given two sets of executed nodes, $N^{ex1} \subset N^{ex2}$, then N^{ex2} describes a point in the execution history that is later than that of N^{ex1} (i.e., the largest timestamp t associated with any node in N^{ex1} will be strictly less than that t' associated with N^{ex2}. However, it could be possible that after

N^{ex1} we causal-consistently replay a node that is not in N^{ex2} (e.g., because its execution is completely independent of the nodes in $N^{ex2} \backslash N^{ex1}$). In general, if $N^{ex1} \not\subset N^{ex2}$ and $N^{ex2} \not\subset N^{ex1}$, then there must be a node $n_1 \in N^{ex1}$, $n_1 \notin N^{ex2}$ that can be causal-consistently undone in the next step; and vice versa, there must be a node $n_2 \in N^{ex2}$, $n_2 \notin N^{ex1}$, that can be causal-consistently undone in the next step. This is ensured by the fact that (1) $n_1 \notin N^{ex1} \cap N^{ex2}$, thus n_1 belongs to a branch not executed in N^{ex2} (thus, n_1 is not causal with respect to any node in $N^{ex1} \cap N^{ex2}$, otherwise, n_1 would belong to N^{ex2}); and (2) n_1 is not causal with respect to any node in N^{ex2} (otherwise, n_1 would belong to N^{ex2}). These properties show that we can use N^{ex} to complement t (t is implicit to N^{ex}). In fact, N^{ex} contains enough information to know how developed are all branches in the R-track because an R-track implicitly induces a Hasse diagram.

Roughly speaking, a node n' can be causal-consistently replayed after a node n if n' happened after n and all causal nodes with respect to n' have been already executed. Formally,

$$\begin{aligned} \mathit{first\text{-}after}_{cc}(\mathcal{G}, N^{ex}) = \{n \in N \mid n \in \Sigma \wedge n \notin N^{ex} \wedge \\ \forall\, n' \in N \cap \Sigma, \mathit{time}(n') \neq \mathit{time}(n), n' \rightsquigarrow n \,.\, n' \in N^{ex}\} \end{aligned}$$

The definition for $\mathit{last}_{cc}(\mathcal{G}, t)$ is analogous:

$$\begin{aligned} \mathit{last\text{-}before}_{cc}(\mathcal{G}, N^{ex}) = \{n \in N^{ex} \mid n \in \Sigma \wedge \\ \nexists\, n' \in N^{ex} \cap \Sigma, \mathit{time}(n') \neq \mathit{time}(n) \,.\, n \rightsquigarrow n'\} \end{aligned}$$

We can also define a causal-consistent version of function *stepNodes*:

$$\begin{aligned} \mathit{stepNodes}_{cc}(\mathcal{G}, N^{ex}, n) = \mathit{causalNodes}(\mathcal{G}, n) \backslash \{n' \in N \mid \\ (n' \mapsto n'') \in A_c^* \wedge n'' \in \Sigma \wedge n'' \in N^{ex}\} \end{aligned}$$

We are now in a position to define a causal-consistent semantics:

($\mathit{Forward}_{cc}$)

$$\frac{n \in \mathit{first\text{-}after}_{cc}(\mathcal{G}, N^{ex}) \quad P = \{\mathit{pos}(n') \mid n' \in \mathit{stepNodes}_{cc}(n)\}}{(\mathcal{G}, N^{ex}) \xrightarrow{P} (\mathcal{G}, N^{ex} \cup \{n\} \cup \{n_s \mid (n \leftrightarrow n_s) \in A_s\})}$$

($\mathit{Backward}_{cc}$)

$$\frac{n \in \mathit{last\text{-}before}_{cc}(\mathcal{G}, N^{ex}) \quad P = \{\mathit{pos}(n') \mid n' \in \mathit{stepNodes}_{cc}(n)\}}{(\mathcal{G}, N^{ex}) \xleftarrow{P} (\mathcal{G}, N^{ex} \setminus (\{n\} \cup \{n_s \mid (n \leftrightarrow n_s) \in A_s\}))}$$

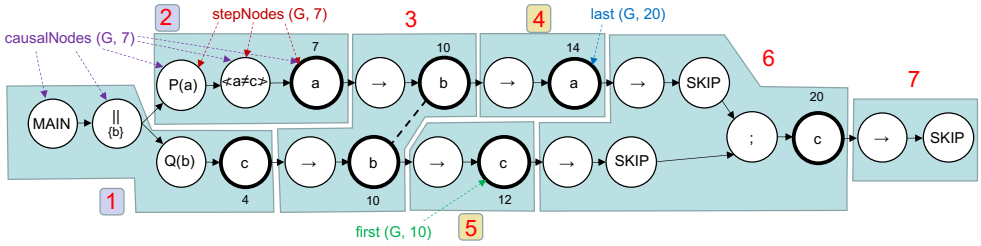


FIGURE 5.5: An R-track from the CSP program described in Example 5.8 produced by the trace ⟨cabcac⟩.

Example 5.8 (Applying functions to a CSP trace)**.**
Consider the following CSP program:

```
channel a,b

MAIN = P(a) ||{b} Q(b) ; c -> SKIP
P(x) = (x → b → a → SKIP)⧏x ≠ c⧐(b → SKIP)
Q(x) = c → x → c → SKIP
```

and the R-track produced by the trace ⟨cabcac⟩, which can be seen in Fig. 5.5.

For the sake of clarity, the R-track does not contain program positions, and it only shows the timestamp of the event nodes (those in bold). Every area with a set of nodes represent a graph rewriting step performed by the semantics, which often corresponds to the occurrence of one event in that area. All the nodes inside an area correspond to the part of the program that induce this step (thus they are actions that can be done or undone). The steps have been numbered from 1 to 7. Steps 1 and 2, and steps 4 and 5, are mutually causal-consistent, and they could be executed in any order. For instance, if we change steps 4 and 5, then the trace would be ⟨cabacc⟩, which is a feasible trace. Similarly, if we change steps 1 and 2, then the trace would be ⟨acbcac⟩, which is another feasible trace. Note, however, that in this case, the semantics would change the nodes that belong to the areas (i.e, the *stepNodes*): the first two nodes would become *stepNodes* of event *a* (node 7) and would not be *stepNodes* of event *c* (node 4). Examples of functions *first-after*, *last-before*, *causalNodes*, and *stepNodes* are also shown.

We end with a theorem that proves the symmetric property of the forward and backward transitions. It is often known as the *loop* lemma [67].

Theorem 5.3 (Loop). *For any forward transition $R \xrightarrow{P} S$, there exists a backward transition $S \xleftarrow{P} R$ and conversely.*

Proof. Assume that $R = (\mathcal{G}, N_1^{ex}) \xrightarrow{P} (\mathcal{G}, N_2^{ex}) = S$. Then, by definition, there exists a node $n \in \textit{first-after}_{cc}(\mathcal{G}, N_1^{ex})$ such that

- $P = \{pos(n') \mid n' \in \textit{stepNodes}_{cc}(n)\}$, and
- $N_2^{ex} = N_1^{ex} \cup \{n\} \cup SYNC_n$.

Recall that $SYNC_n = \{n_s \mid (n \leftrightarrow n_s) \in A_s\}$. Since $n \in \textit{first-after}_{cc}(\mathcal{G}, N_1^{ex})$, we have that

1. $n \in N \cap \Sigma$,
2. $n \notin N_1^{ex}$, and
3. $\forall n' \in N, n' \in \Sigma, time(n') \neq time(n), n' \rightsquigarrow n \,.\, n' \in N_1^{ex}$.

For $S \xleftarrow{P} R$, it suffices to show that 4) $n \in \textit{last-before}_{cc}(\mathcal{G}, N_2^{ex})$, and 5) $N_1^{ex} = N_2^{ex} \setminus (\{n\} \cup SYNC_n)$.

- We first prove that 4) $n \in \textit{last-before}_{cc}(\mathcal{G}, N_2^{ex})$. To this end, we show that $n \in N_2^{ex}$ and $\nexists n' \in N_2^{ex}, n' \in \Sigma, time(n') \neq time(n) \,.\, n \rightsquigarrow n'$. It follows from the definition of $\textit{first-after}_{cc}$ that $\textit{first-after}_{cc}(\mathcal{G}, N_1^{ex}) \subseteq N_2^{ex}$, and hence, $n \in N_2^{ex}$ by the construction of N_2^{ex}. Assume that there exists some $n' \in N_2^{ex}$ such that $n' \in \Sigma, time(n') \neq time(n)$, and $n \rightsquigarrow n'$. Since $n' \in N_2^{ex} = N_1^{ex} \cup \{n\} \cup SYNC_n$, we make a case analysis depending on where n' belongs to.

– Case that $n' = \{n\} \cup SYNC_n$. In this case, we have that $time(n') = time(n)$, which is a contradiction with $time(n') \neq time(n)$.
– Case that $n' \in N_1^{ex}$. In this case, n could not be the first (the first would be n') which is a contradiction with $n \in \textit{first-after}(N_1^{ex})$.

Hence, $n' \notin N_2^{ex}$. This contradicts that $n' \in N_2^{ex}$.

- Next, we prove that 5) $N_1^{ex} = N_2^{ex} \setminus (\{n\} \cup SYNC_n)$. By the definition of N_2^{ex}, we have that $N_2^{ex} \setminus (\{n\} \cup SYNC_n) = N_1^{ex} \setminus (\{n\} \cup SYNC_n)$. It follows from 2) that $N_1^{ex} \setminus (\{n\} \cup SYNC_n) = N_1^{ex} \setminus SYNC_n$. Thus, it suffices to show that

$$N_1^{ex} = N_1^{ex} \setminus SYNC_n.$$

It is trival that $N_1^{ex} \supseteq N_1^{ex} \setminus SYNC_n$. For $N_1^{ex} \subseteq N_1^{ex} \setminus SYNC_n$, we show that $N_1^{ex} \cap SYNC_n = \emptyset$. Assume that there exists some $n_s \in N_1^{ex}$ such that $(n \leftrightarrow n_s) \in A_s$. Since $N_1^{ex} \subseteq N_2^{ex}$, we have that $n_s \in N_2^{ex}$. Then, we have that

$(n, n_s) \in A^*$, and hence $n \rightsquigarrow n_s$. Since $n \notin N_1^{ex}$, we have that $n_s \neq n$, and hence $time(n_s) \neq time(n)$. It follows 4) that

$$\nexists\, n' \in N_2^{ex} \cap \Sigma, time(n') \neq time(n)\ .\ n \rightsquigarrow n'$$

and hence, $n \not\rightsquigarrow n_s$. This contradicts the fact that $n \rightsquigarrow n_s$.

Therefore, by definition, we have that $S \overset{P}{\leftarrow} R$.

The converse can be proved analogously. □

5.3 Implementation and empirical evaluation

The formalization presented in this chapter is an event-based calculus where every step of the defined semantics goes forward or backward in the R-track until the next external event. This means that a single step of that semantics can result in an arbitrary number of rewriting steps (those that fire internal events) in the standard operational semantics [97].

Our implementation, however, strictly follows the standard operational semantics, and thus, it can perform (forward or backward) standard steps (e.g., internal choice) where external events are not necessarily involved. Therefore, the implementation is more general than the theoretical setting, and it goes into the details of handling internal events, and all rules of the standard semantics. However, this does not mean that the implementation must necessarily perform standard steps because the formalization presented has been also implemented, and thus the user can decide to perform (forward or backward) steps based on external events (as in the theoretical setting), any events (external and internal) or any single step of the standard semantics. Hence, the granularity level of the execution can be parametrised.

A practical description of the implementation, called *reverCSP*, is available on Chapter 11. The main bulk of the execution (track generation) is performed by *csp_tracker*, a submodule that executes CSP by creating a coordinating process that handles the track and one process for the first process, typically `MAIN`. The latter derives the body of the process until it ends or is deadlocked. Whenever a parallel operator is found during the execution, two processes are created, and the now parent process is left to continue its execution if a sequential composition requires it. Thus, the number of active processes is roughly equal to the number of CSP processes. The scheduling is left to the Erlang VM, and is not tuned by csp_tracker in any specific direction.

When a backwards step is taken, reverCSP traverses the R-track to reach the appropriate state and display it to the user. When a forwards step is taken

that is present in the graph, reverCSP traverses the graph instead of re-running the simulation.

The rest of this section provides technical details about the implementation so that researchers or developers that want to implement this technique (e.g., for other language) can see the rationale and algorithms used. We use a functional language to describe the implementation.

Given a computation, i.e., an R-track $\mathcal{G} = (N, A_c, A_s)$, the set of reversible actions can be obtained with the following function call: $\mathit{reversible}(\mathit{Leaves}, \emptyset, \emptyset)$ where $\mathit{Leaves} = \{n \mid n \in N, \nexists\, n' \in N \,.\, (n \mapsto n') \in A_c\}$, which is defined in Figure 5.6, Equation 5.1.

All parameters of function *reversible* are subsets of N: L is the set to be analysed, R is an accumulator parameter with the reversible actions (the final output), and C is a list of candidates that should wait for the rest of nodes that are synchronised with them. Function *irp* (is relevant point) should return *true* or *false* whether the node is relevant to stop or not. For instance, for an event node it should return *true*, while for a node of a parallelism operator, it should return *false*. Function *reversible* returns a set of sets, where each set represents an undoable action and all the nodes involved in that action.

The result of function *reversible* can be used to undo one of the undoable actions. The function that removes an action *Sel* from an R-track $\mathcal{G}$ is defined as follows:

$$\mathit{remove}(\mathcal{G} = (N, A_c, A_s), \mathit{Sel}) = (N', A'_c, A'_s)$$

$$\begin{aligned}
\text{where } N' = N \backslash (& \bigcup_{n \in \mathit{Sel}} \{n\} \cup \{n' \mid n' \in N, (n \mapsto n') \in A_c^*\} \cup \\
& \{n' \mid n' \in N, (n \leftrightarrow n') \in A_s^*\}) \\
A'_c = A_c \backslash & \{(n \mapsto n') \mid (n \mapsto n') \in A_c, n \notin N' \wedge n' \notin N'\} \\
& \text{in the following denoted as: } A'_c = A_c|_{N'} \\
A'_s = A_s \backslash & \{(n \leftrightarrow n') \mid (n \leftrightarrow n') \in A_s, n \notin N' \wedge n' \notin N'\} \\
& \text{in the following denoted as: } A'_s = A_s|_{N'}
\end{aligned}$$

When an action is undone with the *remove* function we have a new R-track from which it must be possible to perform a new action (i.e., step forward). Therefore, we need a mechanism to regenerate the state of the tracking semantics given this new R-track. In this way, we can continue the computation from the concrete state given by this new R-track. This has been implemented using function *sft* (state from R-track). This function uses function $\mathit{der}(s)$, which returns all possible derivations of state s using the tracking semantics. Additionally, function *tt* (R-track in time) is used to calculate the R-track $\mathcal{G}'$ in

$$reversible(L,R,C) = \begin{cases} R & \textbf{if } L = \emptyset \\ reversible(L\backslash\{n\}, R \cup \{Sync \cup \{n\}\}, C\backslash Sync) & \textbf{if} \exists n \in L \wedge irp(n) \wedge Sync \cap C = Sync \\ reversible(L\backslash\{n\}, R, C \cup \{n\}) & \textbf{if } \exists n \in L \wedge irp(n) \wedge Sync \cap C \neq Sync \\ reversible((L \cup Prev)\backslash\{n\}, R, C) & \textbf{if } \exists n \in L \wedge not(irp(n)) \end{cases} \tag{5.1}$$

$$\text{where } Sync = \{n' \mid n' \in N, (n \nleftrightarrow n') \in A_s\}, Prev = \{n' \mid n' \in N, (n' \mapsto n) \in A_c\}$$

$$sft(s,T,t) = \begin{cases} s & \textbf{if } \forall n \in N \,.\, t > time(n) \\ sft(s',T,t+1) & \textbf{if } \exists n \in N \,.\, time(n) \leq t \wedge \exists s \rightarrow s' = (\_,T_c,\_,\_) \in der(s) \,.\, tt(T,t) = T_c \\ sft(s,T,t+1) & \textbf{if } \exists n \in N \,.\, time(n) \leq t \wedge \nexists s \rightarrow s' = (\_,T_c,\_,\_) \in der(s) \,.\, tt(T,t) = T_c \end{cases} \tag{5.2}$$

$$tdrs(n, T = (N, A_c, A_s)) = \begin{cases} (pos, \bullet) & \textbf{if } n = \bullet_{pos} \\ (next(n), n) & \textbf{if } n \notin \{||, |||\} \wedge \nexists n' \in N.(n \mapsto n') \in A_c \\ tdrs(n', T) & \textbf{if } n \notin \{||, |||\} \wedge \exists n' \in N.((n \mapsto n') \in A_c \\ & \quad \wedge \nexists n'' \in N.(n \mapsto n'') \in A_c \wedge n'' \neq n') \\ (n'_l \; pos(n)_{(pos(n),p'_l,p'_r)} \; n'_r, n) & \textbf{if } n \in \{||, |||\} \end{cases} \tag{5.3}$$

where $(n_l, n_r) = lr(n,T) \wedge (n'_l, p_l) = tdrs(n_l, T) \wedge (n'_r, p_r) = tdrs(n_r, T) \wedge ((p'_l = n \wedge p_l = \bullet) \wedge (p'_l = p_l \wedge p_l \neq \bullet)) \wedge ((p'_r = n \wedge p_r = \bullet) \wedge (p'_r = p_r \wedge p_r \neq \bullet))$, and function $next(n)$ returns the expression that follows the expression of node n:

$$next(n) = \begin{cases} pos(n).2 & \textbf{if } n = \; \rightarrow \\ n.\Lambda & \textbf{if } n \in ProcessNames \\ \top & \textbf{if } n = SKIP \\ \bot & \textbf{if } n = STOP \\ pos(n).b & \textbf{if } n = \sqcap.b \end{cases} \tag{5.4}$$

FIGURE 5.6: Definition of multiple functions used throughout the implementation.

$$lr(n,T) = \begin{cases} (n_l, n_r) & \textbf{if } \exists n_l, n_r \in N \,.\, (n \mapsto n_l) \in A_c \\ & \quad \wedge\, (n \mapsto n_r) \in A_c \wedge pos(n_l) = pos(n).1 \\ & \quad \wedge\, pos(n_r) = pos(n).2 \\ (n_l, \bullet_{pos(n).2}) & \textbf{if } \exists n_l \in N \,.\, ((n \mapsto n_l) \in A_c \\ & \quad \wedge\, pos(n_l) = pos(n).1) \\ & \quad \wedge\ \nexists n_r \in N \,.\, ((n \mapsto n_r) \in A_c \\ & \quad \wedge\, pos(n_r) = pos(n).2) \\ (\bullet_{pos(n).1}, n_r) & \textbf{if } \nexists n_l \in N \,.\, ((n \mapsto n_l) \in A_c \\ & \quad \wedge\, pos(n_l) = pos(n).1) \\ & \quad \wedge\, \exists n_r \in N \,.\, ((n \mapsto n_r) \in A_c \\ & \quad \wedge\, pos(n_r) = pos(n).2) \\ (\bullet_{pos(n).1}, \bullet_{pos(n).2}) & \textbf{if } \nexists n_l \in N \,.\, ((n \mapsto n_l) \in A_c \\ & \quad \wedge\, pos(n_l) = pos(n).1) \\ & \quad \wedge\ \nexists n_r \in N \,.\, ((n \mapsto n_r) \in A_c \\ & \quad \wedge\, pos(n_r) = pos(n).2) \end{cases} \tag{5.5}$$

$$\text{where } T = (N, A_c, A_s)$$

FIGURE 5.6: (Cont.) Definition of multiple functions used throughout the implementation.

a given time t using the R-track $\mathcal{G}$ of the whole computation. Its definition is the following:

$$tt(\mathcal{G} = (N, A_c, A_s), t) = (N', A'_c, A'_s) = \mathcal{G}'$$

$$\begin{aligned} \text{where } N' &= \{n \mid n \in N \,.\, time(n) \leq t\} \\ A'_c &= A_c|_{N'} \\ A'_s &= A_s|_{N'} \end{aligned}$$

We can now define function *sft*, which allows us to start the computation in the same point where the computation associated with the R-track stopped, as can be seen in Figure 5.6, Equations 5.2, 5.3 and 5.4.

Finally, function *lr* (Figure 5.6, Equation 5.5) returns the left and right nodes of a bifurcation operator. In case any (or both) of the branches was not unfolded, the symbol • is used to denote this fact. The • is labelled with the first position of the non-evaluated branch.

TABLE 5.1: Size of the tracks generated with a given runtime

Benchmark	Runtime (ms)	#Nodes	#Arcs
ABP.csp	[2208.16 2209.25 2210.34]	[1505.61 1506.17 1506.73]	[1303.10 1303.63 1304.17]
ATM.csp	[630.17 690.18 750.19]	[364.09 405.64 447.19]	[300.74 334.67 368.61]
Buses.csp	[126.40 127.19 127.97]	[22.00 22.00 22.00]	[18.00 18.00 18.00]
CPU.csp	[189.97 190.74 191.51]	[87.43 87.76 88.09]	[71.23 71.50 71.77]
Disk.csp	[209.07 210.10 211.13]	[148.50 148.74 148.98]	[123.59 123.78 123.78]
Loop.csp	[2133.02 2133.99 2134.96]	[1537.53 1538.34 1539.14]	[1230.05 1230.69 1231.35]
Oven.csp	[238.64 241.92 245.20]	[157.16 163.37 169.59]	[162.68 169.33 175.98]
ProdCons.csp	[2134.59 2135.43 2136.27]	[1535.43 1536.09 1536.75]	[1228.08 1228.61 1229.15]
ReadWrite.csp	[2148.76 2149.71 2150.65]	[1475.85 1476.56 1477.28]	[1252.47 1253.34 1254.22]
Traffic.csp	[165.34 166.35 167.36]	[61.18 64.37 67.56]	[47.73 50.13 52.53]
Average	[1018.41 1025.49 1019.76]	[689.478 694.90 700.33]	[573.77 578.37 582.98]

5.3.1 Empirical evaluation

In order to precisely quantify the performance of ReverCSP, we conducted several experiments to evaluate the size and the time needed to produce tracks. For the evaluation, a set of heterogeneous benchmarks was selected from public CSP repositories. All of them have been previously used to evaluate other CSP techniques and tools. The selected benchmarks ensure that they cover all the CSP syntax, and also that they have a wide range of possible executions (finite executions and infinite executions with both finite and infinite nested parallelism). Each benchmark includes a header describing it and providing details about the authors and a reference to its source.[1]

The empirical evaluation strictly followed the method and guidelines proposed in [41, 101]. The same hardware configuration was used to assess all benchmarks: Intel® Xeon® E5504 Processor (2 Ghz, 4MB Cache, 4 cores) with 16GB RAM. In order to avoid interference of other programs, all processes except ReverCSP were stopped while the benchmarks were running. Every benchmark was run 1001 times. In all cases, the first iteration was discarded to avoid the influence of cached data persisting in the disk, or dynamically loaded libraries stored in physical memory. Thus, we finally obtained 1000 statistical values for each benchmark. This process was repeated for the 10 benchmarks. In order to study the effect of statistical dispersion, we computed both the harmonic and the arithmetic mean. The former was sufficiently low so that we could use the arithmetic mean in our table results.

[1] Benchmarks are available at `https://github.com/mistupv/csp_tracker/tree/master/benchmarks`.

Because the benchmarks could produce infinite executions (e.g., due to livelock processes) we automatically stopped them after a timeout of 2 seconds. Thanks to the good scalability of the tool, this threshold was enough to produce long tracks with many parallel processes and more than 1500 nodes. This threshold allowed us to compare the track produced by different CSP specifications (with different number of synchronizations and parallel processes, and different levels of complexity, etc.) executed exactly the same time. The results are shown in Table 5.1.

In the table, we use symmetric 0.99 confidence intervals. The meaning of the columns is the following: *Benchmark* is the name of the benchmark. *Runtime* is the time that the benchmark was executed before it eventually ended or before it was stopped. *#Nodes* is the number of nodes that compose the track. *#Arcs* is the number of (control and synchronization) arcs that compose the track.

5.3.2 Study of the time and memory overhead

The generation of R-tracks introduces a measurable overhead in both time and memory, as opposed to simulating CSP without storing any state information apart from the necessary to run the following step. To determine the scale of this overhead, we conducted another empirical evaluation, measuring the time and memory used.

The benchmark suite used is very similar to the one used in the performance evaluation, except for a small change in *Buses* and the removal of *ATM* and *ABP*, and can be found alongside with the code and appropriate compilation flags to reproduce the evaluation in the git tags `TDPS_bench_time` and `TDPS_bench_memory` in our repository (`https://github.com/mistupv/csp_tracker`). The same hardware configuration was used to assess all benchmarks: Intel® Xeon® E3-1220 v3 @ 3.10GHz Processor with 8GB DDR3 RAM at 1333MHz (single channel). In order to avoid interference from other programs, all other non-essential processes were stopped.

Two modes of execution are present in the code: `run` and `track`. The latter is the one used to generate R-tracks, which can be then interactively traversed by the user. The former is very similar, but skips all possible track-generating operations, and simulates a CSP interpreter that executes a CSP specification as fast as possible.

Each time benchmark was run 1001 times in each mode: `run` and `track`. The same Erlang VM was used for each block of 1001 benchmarks, but the first iteration was discarded to avoid the delay introduced by loading the required code and libraries from disk to RAM.

TABLE 5.2: Memory and time consumption overhead introduced by R-tracks

Benchmark	kilobytes per operation		Memory	microseconds per operation		Time
	`run`	`track`	overhead	`run`	`track`	overhead
Buses.csp	$0.26 \pm 0.80\%$	$36.28 \pm 1.19\%$	140.375	$37.62 \pm 0.22\%$	$2312.07 \pm 0.04\%$	61.466
CPU.csp	$11.38 \pm 0.63\%$	$38.51 \pm 1.23\%$	3.384	$22.86 \pm 2.10\%$	$117.63 \pm 1.18\%$	5.146
Disk.csp	$8.83 \pm 0.52\%$	$45.81 \pm 1.85\%$	5.188	$24.63 \pm 1.50\%$	$171.01 \pm 0.79\%$	6.944
Loop.csp	$0.04 \pm 1.33\%$	$33.83 \pm 0.85\%$	930.505	$12.50 \pm 0.19\%$	$1314.21 \pm 0.06\%$	105.171
Oven.csp	$13.77 \pm 0.77\%$	$76.86 \pm 1.88\%$	5.580	$30.66 \pm 1.84\%$	$344.66 \pm 4.43\%$	11.242
ProdCons.csp	$0.04 \pm 1.87\%$	$33.24 \pm 0.86\%$	890.739	$15.03 \pm 0.04\%$	$1326.11 \pm 0.09\%$	88.215
ReadWrite.csp	$13.78 \pm 1.04\%$	$69.94 \pm 0.97\%$	5.077	$707.12 \pm 0.32\%$	$2492.11 \pm 0.31\%$	3.524
Traffic.csp	$37.92 \pm 1.46\%$	$96.18 \pm 1.57\%$	2.536	$53.12 \pm 2.72\%$	$163.20 \pm 2.56\%$	3.072

Each memory benchmark was run 1000 times in each of the aforementioned modes. In this case, each iteration was run in its own Erlang VM, as to guarantee that no remnants of the previous iteration lived on in memory. Before each iteration, the necessary code and libraries were manually loaded as to obtain the most precise measurement possible.

Thus, we obtained 1000 statistical values per benchmark and mode for each metric (time and memory consumption). A timeout of one second was established, as some processes are infinitely long, and would not stop otherwise. As a way to compare both modes of execution, we measured the number of operations that each process took, so that the `track` and `run` versions could be easily compared. Our results are shown in Table 5.2.

In the table, we show the 0.99 error margins as a percentage. The first column is the name of the file; the second and third show the consumption of *kilobytes per operation* taken, for both `run` and `track` modes. *Memory overhead* displays the ratio, showing that generating R-tracks is several times more memory-intensive than running the CSP specification. *Microseconds per operation* shows the average time needed per operation for each mode, and *time overhead* displays the ratio, with meaning similar to *memory overhead*.

As can be seen, there is an overhead of up to an order of magnitude in general, but three of the benchmarks have overheads of two orders of magnitude for time and three for memory. These three (`Buses`, `Loop`, and `ProdCons`) are not only infinite but have an ever-increasing number of active processes, which increases the size of the R-track considerably. The other benchmarks keep the number of active processes stable, even if they do run indefinitely.

5.4 Related Work

Reversibility [1] is the ability of a program to be executed both forward and backward, thus having the possibility to go back to past states. This ability has been studied in most languages (e.g., CSS [20], π-calculus [66], Erlang [69], and CSP [38], among many others) and has different applications such as program comprehension and debugging [42], quantum computing [6], biological systems modelling [12], etc.

In the specific case of debugging, different approaches have been defined for rollback-recovery [22, 43], whose most simple instance is the *undo* button, which restores the immediate previous state. While reversibility in a sequential system is understood as the reverse of a set of actions in the opposite order as they were done (see, e.g., [84]), reversibility in a concurrent system is not intuitive at all, because the order of actions is often undefined (e.g., many actions can happen concurrently). For this reason, reversibility is particularly useful to debug concurrent languages, because there is no guarantee that a bug that appears in the original computation is replayed inside the debugger. This problem is usually tackled by so-called replay debugging [68]. Our replay debugger uses a new data structure called *track* that allows us to reconstruct all states in the execution in both directions. Tracks were introduced in [76] as a means to record executions. The original idea of that paper is that they could be used for program comprehension and tracing. The idea of using them for reversibility was proposed in [38]. Here, we extend the work done in [38] with several new ideas and a formal theory about how to use R-tracks for reversibility. This theory includes a formal reversibility semantics for CSP. With that purpose, we have slightly extended the original notion of tracks so that they can be used to replay computations forward and backward. This new track is called R-track. The reversible semantics proposed is completely novel with respect to [76]. The new implementation is conservative with respect to the original tracks because it can generate standard CSP tracks, which are useful for, e.g., program comprehension; but it also generates the extended tracks proposed here. With this extension our debugger can replay the execution in any causal-consistent order.

An R-track defines a partial order through the causality relation $n_1 \rightsquigarrow n_2$, which is reflexive, antisymmetric, and transitive. Therefore, the model proposed based on R-tracks is a model for true concurrency based on a non-interleaving semantics which explicitly represents causality in the R-track. Other semantics for reversibility that are related to our work are the non-interleaving semantics for CSS [20] and π-calculus [21].

A system that is similar to ours, but for Erlang, and using a different notion of tracks, is CauDEr, a causal-consistent reversible debugger for Erlang [70].

The functionality of CauDEr is similar to our tool because it allows undoing actions in a causal-consistent manner. Causal-consistent reversibility was introduced in [18], with RCCS, a reversible calculus for CCS. It introduced a mechanism to attach memories to threads to keep history information. Later interesting approaches are [42, 43]. A survey that very nicely explains the evolution of this field can be found in [67].

Other approaches that are related to our work are [9, 11] and [68]. First, in [9], the authors introduced a modular framework for defining causal-consistent reversible extensions of concurrent models and languages. This work has inspired some of our ideas to define the extended tracks. The work by Brown and Sabry [11] is another interesting work that also proposes two semantic reversible models for a CSP-based language embedded in Scala. They also report that their implementation was evaluated and showed that a practical reversible distributed language can be efficiently implemented in a fully distributed manner. Unfortunately, the implementation is not publicly available. Finally, the work by Lanese et al. [68] proposes a novel approach for replay debugging that is called *controlled causal-consistent replay*. It is called "controlled" in the sense that the debugger shows all and only the causes of an error. This approach is also related to causally-consistent dynamic slicing [83]. However, while our approach uses tracks to traverse the computation forward and backward, dynamic slicing uses a trace to extract the part of the code (the so-called slice) that could influence a given behaviour. Another difference is the target language: pi calculus vs CSP.

Part III

Data Dependence

Chapter 6

Modelling Context-Sensitive, Shared-Memory Data Dependence

> Once you've met someone, you never really forget them
>
> *Spirited Away* (2001)

As we explained in Section 2.4, the SDG can be used to represent the sequential dependences of a program, thus computing valid and precise slices of sequential, multi-procedure programs. However, shared-memory concurrency introduces complexities, which need to be reflected on the dependence graphs.

Currently, the most accurate approach for shared-memory concurrent programs is the one by Nanda et al. [79], who defined a new graph called *threaded System Dependence Graph* (tSDG) and a new slicing algorithm that produces complete context-sensitive slices. The tSDG effectively and elegantly detects impossible executions (the so-called 'time travel' problem), obtaining more precise slices than their predecessors.

Nevertheless, in this chapter we show that the problem has not been solved yet. Specifically, we show that the tSDG produces inaccurate slices in different situations because it is not always context-sensitive (see Section 2.3 for an explanation of the concept). Example 6.1 shows a counterexample designed to illustrate the inaccuracy of the tSDG.

Example 6.1 (Inaccurate slice produced by the tSDG)**.**
Consider the program in Fig. 6.1a, where a thread `T0` launches two threads `T1` and `T2` that are executed in parallel. Consider also variable `a` in line 7 as the slicing criterion (represented as $\langle 7, a \rangle$). A program slicer can determine

```
1  // Thread T0
2  a = 1;
3  call f(2);
4  cobegin{ // Thread T1
5    call f(3);
6    call f(4);
7    print(a);
8  }{ // Thread T2
9    call f(5);
10 }
11
12 procedure f(x)
13   a = x;
```

(A) Expected slice

```
1  // Thread T0
2  a = 1;
3  call f(2);
4  cobegin{ // Thread T1
5    call f(3);
6    call f(4);
7    print(a);
8  }{ // Thread T2
9    call f(5);
10 }
11
12 procedure f(x)
13   a = x;
```

(B) Old tSDG slice

FIGURE 6.1: Minimal slice (left) and old tSDG ([79]) xslice computed (right) w.r.t. slicing criterion $\langle 7, a \rangle$.

what parts of the code can influence the slicing criterion: the value of variable `a` in line 7 (`T1`) may come from the definition of `a` in line 13 executed (i) during the call `f(4)` in line 6 (`T1`); or (ii) during the call `f(5)` in line 9 (`T2`) (depending on which was executed last). Therefore, the slice (in black) does not contain lines 2, 3, and 5 (in gray) because they cannot influence the slicing criterion. This result can only be obtained by a *context-sensitive* analysis. A *context-insensitive* analysis would include lines 3 and 5 in the slice because they can influence lines 12 and 13 and, thus, the slicing criterion.

The slice in Fig. 6.1b has been computed with a tSDG. In this slice, two calls to procedure `f` (lines 6 and 9) are correctly included in the slice together with the code in procedure `f`; but the calls to procedure `f` in lines 3 and 5 in `T0` and `T1` respectively, are unnecesarily included in the slice. As explained before, even though these calls actually define variable `a`, their values are always overwritten before the execution of the slicing criterion (either by the call in line 6 or the one in line 9) preventing them from influencing it, but the tSDG includes them as a context-insensitive analysis would do.

Example 6.1 is our first contribution: a counterexample that reveals the imprecision of the tSDG. It is not always context-sensitive and may unnecessarily include in the slice procedure calls that cannot influence the slicing criterion. In this work we propose an alternative graph model (a new tSDG) that solves this imprecision. It is worth to remark that, despite receiving the same name, our tSDG is not a modification over Nanda et al.'s tSDG, but a new graph built from the SDG of a program. To build our new representation, we take advantage of the good properties and principles of the tSDG, but we

change the way in which *interference* dependence is handled. In the new tSDG, interference dependences are represented using the same principle of summary arcs in the SDG: the idea is to represent all interference dependences that connect statements in different procedures through their call statements, thus making all interference dependences intraprocedural (as it happens with flow dependence). This transformation allows us to use the standard two-phase SDGs algorithm by Horwitz et al. [54], enhanced with Nanda et al.'s approach to avoid time travel.

6.1 Background

This section presents the standard program representations used to slice concurrent programs, based on the SDG presented on Chapter 2. We consider concurrent programs with shared memory. Threads interact accessing common variables and there is no other synchronization mechanism between threads. Threads are created via the classical `cobegin` language construct (see Fig. 6.1).

The threaded version of the CFG is called *threaded Control-Flow Graph* (tCFG). In the tCFG the beginning and end of each thread in `cobegin` blocks are modelled with two extra nodes labelled as *Start* and *End*, analogously to *Enter* and *Exit* nodes in CFGs, embedded in the surrounding CFG of the procedure that contains them.

The concurrent counterpart of the PDG is the *threaded Program Dependence Graph* (tPDG). The tPDG contains the same nodes as the tCFG after removing not only the *Exit* node, but also the *End* nodes of all threads. As the PDG, the tPDG includes control and data arcs, but it also includes additional arcs that represent a new type of dependence exclusive to concurrent programs called *interference dependence*.

Definition 6.1 (Interference Dependence [79])**.** *A node n is* interference dependent *on a node m if m defines a variable v, n uses the variable v, and n and m can execute in parallel.*

Interference dependence differs from control and flow dependences because it is intransitive. If a node n_3 is interference dependent on a node n_2 and, in turn, n_2 is interference dependent on a node n_1, n_3 is interference dependent on n_1 only if there is a possible execution sequence $\langle n_1, n_2, n_3 \rangle$. This is detected by using the tCFG. All feasible execution sequences are called *threaded witnesses*.

Definition 6.2 (Threaded witness [79])**.** *Given a tCFG G of a concurrent program, a threaded witness is an ordered sequence of nodes* $\langle n_1, n_2, \ldots, n_k \rangle$ *belonging to G*

```
1  // Thread T0
2  a = 2;
3  cobegin {
4    // Thread T1
5    b = a;
6  }{
7    // Thread T2
8    b = 5;
9    a = b;
10   a = 10;
11 }
```

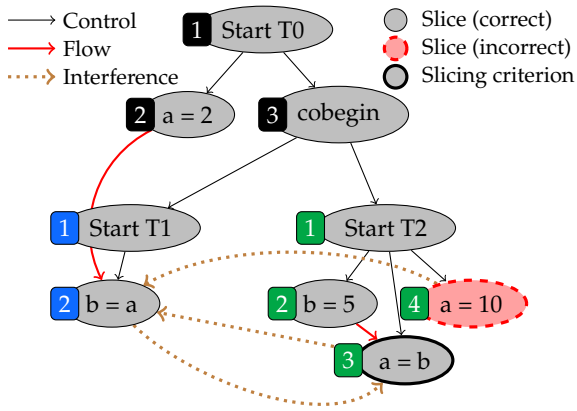


FIGURE 6.2: Concurrent program with time travel interferences and its corresponding tSDG.

such that any subsequence of nodes $n_{i_1}, n_{i_2}, \ldots, n_{i_j}$, $j \leq k$, *belonging to the same thread,* T_i, *forms a realizable path in* T_i.

Intuitively, a threaded witness can be interpreted as a sequence of tCFG nodes that form a valid execution chronology.

When we slice concurrent programs, two main problems arise: the *time travel problem* and the incompleteness problem produced by the use of *summary arcs computed for sequential programs*. In the following subsections, we explain the nature of both problems and describe the solution provided by Nanda et al. [79] (the tSDG) to obtain context-sensitive slices.

6.1.1 The time travel problem

The time travel problem consists in determining whether a dependency chain is chronologically feasible. If it is not solved, statements may be included in the slice even when they cannot affect the slicing criterion because they cannot be executed before it. This situation occurs when we slice concurrent programs with the standard slicing algorithm: interference arcs are transitively traversed during the slicing process, moving back and forth from the current thread to another. Because the traversal does not store information about the sequence of nodes traversed, we may return to a statement of the thread that cannot be executed before the slicing criterion, obtaining an execution sequence that is not an interprocedural threaded witness according to Definition 6.2.

Example 6.2 (Time travelling with interference dependences)**.**
Consider the fragment of code in Fig. 6.2, where threads `T1` and `T2` are executed in parallel. Its associated tSDG is shown on the right (during this example, pay no attention to the coloured numbers next to each node). Consider also variable `a` in line 9 as the slicing criterion $\langle 9, a \rangle$, which takes its value from `b`. In turn, `b` can be defined either in statement 8 in `T2` or in statement 5 in `T1`. If we follow the interference arc to consider the second case, we reach `b = a` in line 5 of `T1`. `a`'s value can come from three different program points (the three incoming flow/interference arcs):

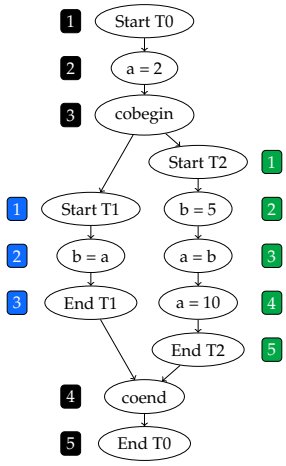
(i) `a` might be flow dependent on statement 2 in `T0`,
(ii) `a` might be interference dependent on statement 9 in `T2` (the `a` definition of the slicing criterion), and
(iii) `a` might be interference dependent on statement 10 in `T2`.

When node `b = a` is reached, the slicing traversal continues backwards to all three statements. (i) is a definition `a = 2`, which always occurs before executing `T1`, so its inclusion in the slice is correct. (ii) coincides with the slicing criterion and is already part of the slice, so its inclusion has no effect. Finally, (iii) defines `a = 10` in `T2`, but always "in the future", i.e., after executing the slicing criterion (the sequence $\langle 10, 5, 9 \rangle$ is not an interprocedural threaded witness). Therefore, this definition can never affect the value of the slicing criterion and its inclusion in the slice is erroneous.

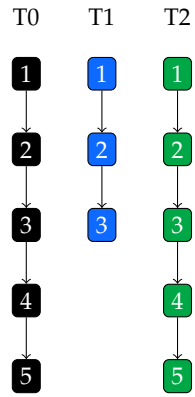
To solve the time travel problem, Nanda et al. [79] proposed a procedure based on the *Topological Number Graph* (TNG), which is an acyclic representation of the tCFG used to check whether a statement can be executed before another one by solving a simple reachability problem. The TNG represents all instants of time that can happen in an execution of a thread and associates to each instant of time the statements that can occur at that time. Each instant of time is represented with a timestamp.

Example 6.3 (Stopping time travel with timestamps)**.**
The tCFG of the code in Fig. 6.2 is shown in Fig. 6.3a. Each node is assigned a single timestamp^a. The corresponding TNG for each thread is shown in Fig. 6.3b.

Timestamps allow us to know, at any time, at which instant of time each thread is being executed. To illustrate how the tSDG solves the time travel problem, we show step by step how the algorithm traverses the graph from the slicing criterion to the statement in line 10 of Fig. 6.2 (this statement should be excluded from the slice). Table 6.1 describes the slicing traversal, logging the following information:



(A) tCFG of the program in Fig. 6.2



(B) TNG of the different threads of the program in Fig. 6.2

FIGURE 6.3: tCFG of the code in Fig. 6.2 and its corresponding TNG for each thread.

Step Order in which the arcs of the graph are traversed.
Last Node The last node visited in the traversal.
Arc Type The type of the arc being traversed in the current step.
New Node The node reached after traversing the current arc. The table lists both the code contained in the node (**Stmt.**) and the timestamps associated with it (**Timestamp**).
Traversal State Lists the timestamp of each thread's last traversed statement (`T0`, `T1`, and `T2`).
Stop? Should the current arc be traversed?

Step 0 represents the start of the traversal. In this step the slicing criterion is the *New Node*, and the corresponding *Traversal State* is updated by annotating `T2` with the set of timestamps $\{3\}$. Step 1 represents the traversal of the interference arc from the slicing criterion to `b`'s definition in `T1`. The *Timestamp* updates the *Traversal State* of `T1` from $\perp$ to $\{2\}$. Finally, Step 2 represents the traversal of the interference arc from the definition of `b` in line 5 to the definition of `a` in line 10 of `T2`. In this step, we detect time travelling: the current timestamp in `T2` is 3, but we have reached a *New Node* with *Timestamp* 4 in `T2`. This means that the current dependencies chain can never be executed (4 is always executed after 3, thus 3 cannot depend on 4). Therefore, the traversal stops, and the new node (a = 10) is correctly excluded from the slice.

[a] Because the code in this example does not contain loops, each statement is trivially assigned to one single timestamp. Later examples show complex timestamps, like a statement

TABLE 6.1: Slicing traversal of the program in Figure 6.2 from the slicing criterion to statement 10

Step	Last Node	Arc Type	New Node		Traversal State			Stop?
			Stmt.	Timestamp	T0	T1	T2	
0	-	-	a = b	$\{3^{T2}\}$	$\perp$	$\perp$	{3}	No
1	a = b	Interference	b = a	$\{2^{T1}\}$	$\perp$	{2}	{3}	No
2	b = a	Interference	a = 10	$\{4^{T2}\}$	$\perp$	{2}	{3}	Yes

being assigned to more than one timestamp.

6.1.2 The sequential summary incompleteness problem

The sequential summary incompleteness problem arises when standard summary arcs (computed for sequential programs) are used to slice concurrent programs. This problem is orthogonal to the time travel problem, and it is explained in Example 6.4.

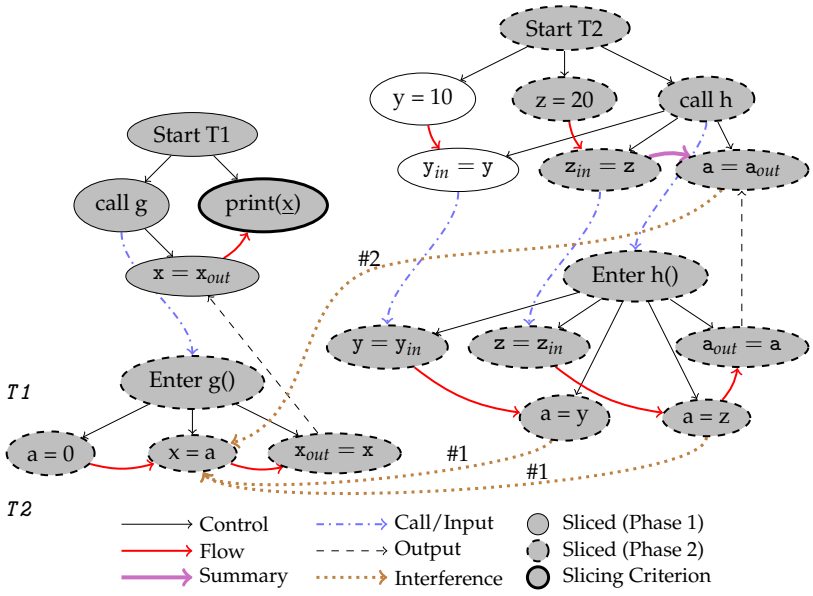
Example 6.4 (The two-pass slicing algorithm is incomplete in the tSDG)**.** Consider the code in Fig. 6.4a. Two threads (`T1` and `T2`) are executed in parallel. `T1` uses variable `a` through its call to procedure `g`. Variable `a` is defined twice in `T2` through its call to procedure `h`.

Figure 6.4b shows the tSDG sliced with the standard SDG algorithm (Algorithm 2), w.r.t. $\langle 4, \texttt{x}\rangle$. We show explicitly the phase in which each node was added to the slice, as that is the source of the problem.

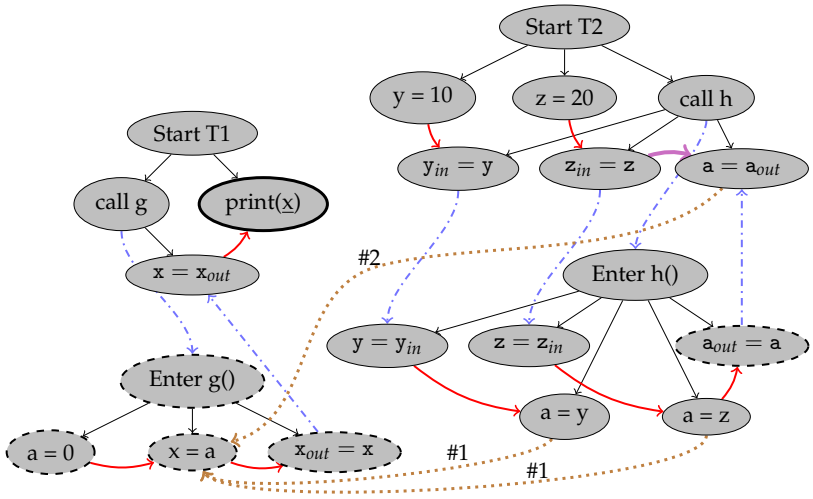
The SDG slice is an incomplete slice. During phase 1, *output* arcs cannot be traversed, thus only *call g* is reached, but not *Enter g*. Then, phase 2 traverses all arcs except *call* and *input* arcs. Therefore, the traversal reaches procedure `g`, including the definition `x = a` in line 14. From this node the algorithm traverses three *interference* arcs. The first two are labelled with #1. We reach the two definitions of `a` inside procedure `h` and their related dependences: both formal-in nodes of variables `y` and `z` are reached. Since we are in phase 2, *input* arcs cannot be traversed and, thus, we do not exit procedure `h`. The third interference arc is labelled with #2 and we reach the actual-out node of `call h` in `T2` ($\texttt{a} = \texttt{a}_{out}$). With the summary arc, we include one of the actual-in nodes ($\texttt{z}_{in} = \texttt{z}$). Unfortunately, actual and formal nodes (and thus, summary arcs) only represent sequential dependences, and not interferences. If we think sequentially, `a`'s value can only come from `a = y`. If we think concurrently, from the point of view of `T1`, `a`'s value can come from either assignment in `h` (`a = y` or `a = z`). Alas, `y` is not

```
1  cobegin {
2    // Thread T1
3    call g();
4    print(x);
5  }{
6    // Thread T2
7    y = 10;
8    z = 20;
9    call h();
10 }
11
12 procedure g
13   a = 0;
14   x = a;
15
16 procedure h
17   a = y;
18   a = z;
```

(A) A program



(B) The tSDG, sliced with the SDG algorithm.



(C) The tSDG, sliced with Nanda et al.'s algorithm.

FIGURE 6.4: The summary incompleteness problem.

considered relevant to this call, and is excluded, along with its definition (`y = 10`), making the slice incomplete.

After detecting this problem, Nanda et al. proposed a modification in the slicing algorithm to solve it. They noticed that the problem arises when an interference arc is traversed during phase 2. But, far from redefining the summary arcs' computation to cope with the interference behaviour, they decided to consider every node reached when traversing an interference arc as a new slicing criterion, restarting the slice from this node in phase 1 even if it was reached in phase 2. A detailed description of the algorithm they proposed can be found in [79]. The result of running their algorithm is shown in Fig. 6.4c.

The tSDG slicing algorithm works in a similar way as the SDG slice, until it encounters interference arcs. When an interference arc has been traversed, the slice returns to phase 1 and, therefore, input arcs can be traversed. In our case, the input arc ($y_{in} = y \rightarrow y = y_{in}$) can be traversed, reaching the actual-in and all its dependences, and correctly including node `y = 10`.

The modifications in the slicing algorithm introduced by Nanda et al. provide a new mechanism to overcome summary limitations with an alternative graph traversal, but do not modify the sequential computation of summary arcs in the graph. Their model ignores the presence of summary arcs in a procedure call when the corresponding procedure definition is reached via interference arcs. Thus, some of the actual-in nodes of the call may be reached by both summary and input arcs.

6.2 Limitations of the current tSDG

Nanda et al.'s new slicing algorithm solves the incompleteness problem, but introduces a problem of imprecision: the tSDG can generate context-insensitive slices, as shown in Example 6.1. Let us explore the algorithm in detail:

Example 6.5 (The tSDG is not context-sensitive)**.**
Consider the program in Fig. 6.1 and its associated tSDG shown in Fig. 6.5. The tSDG marks the nodes included in the slice computed with Nanda et al.'s algorithm with respect to the slicing criterion $\langle 7, \mathtt{a} \rangle$ (the node in bold). The node's outline shows the phase in which they are reached, white nodes are not part of the slice, and red nodes are unnecessarily included in the slice. In the graph, we can see that the slice includes irrelevant calls to procedure `f` and their corresponding actual-in nodes $\mathtt{x}_{in} = 2$ and $\mathtt{x}_{in} = 3$ in the slice. Table 6.2 shows the path followed by the slicing algorithm to reach formal-in node $\mathtt{x} = \mathtt{x}_{in}$ during Phase 1 (which is the cause of including

```
1  // Thread T0
2  a = 1;
3  call f(2);
4  cobegin{ // T1
5    call f(3);
6    call f(4);
7    print(a);
8  }{ // T2
9    call f(5);
10 }
11
12 procedure f(x)
13   a = x;
```

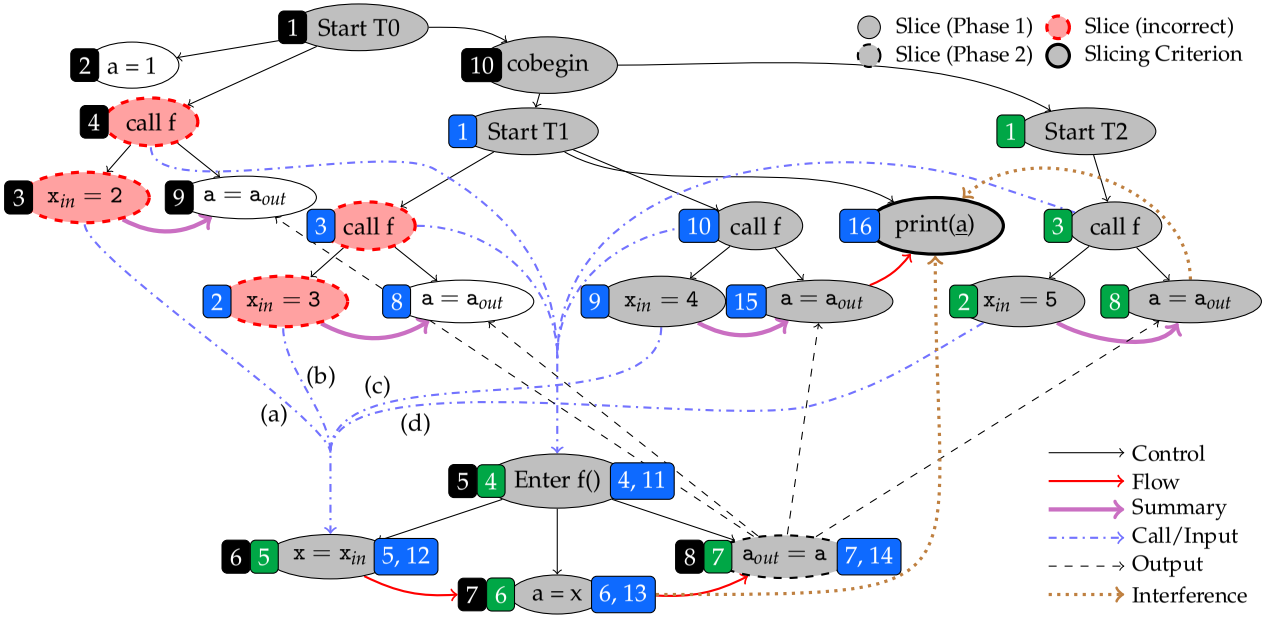


FIGURE 6.5: Nanda et al.'s tSDG of the program in Fig. 6.1.

TABLE 6.2: Slicing traversal of the program in Fig. 6.1 from the slicing criterion $\langle 7, a \rangle$ to all the actual-in nodes $\texttt{x} = \texttt{x}_{in}$.

Step	Last Node	Arc Type	New Node		Traversal State		
			Statement	Timestamps	T0	T1	T2
0	-	-	print(a)	$\{16^{T1}\}$	$\bot$	$\{16\}$	$\bot$
1	print(a)	Interf.	a = x	$\{7^{T0}, 6^{T1}, 13^{T1}, 6^{T2}\}$	$\{7\}$	$\{6, 13\}$	$\{6\}$
2	a = x	Flow	$\texttt{x} = \texttt{x}_{in}$	$\{6^{T0}, 5^{T1}, 12^{T1}, 5^{T2}\}$	$\{6\}$	$\{5, 12\}$	$\{5\}$
3a	$\texttt{x} = \texttt{x}_{in}$	Input	$\texttt{x}_{in} = 2$	$\{3^{T0}\}$	$\{3\}$	$\{5, 12\}$	$\{5\}$
3b	$\texttt{x} = \texttt{x}_{in}$	Input	$\texttt{x}_{in} = 3$	$\{2^{T1}\}$	$\{6\}$	$\{2\}$	$\{5\}$
3c	$\texttt{x} = \texttt{x}_{in}$	Input	$\texttt{x}_{in} = 4$	$\{9^{T1}\}$	$\{6\}$	$\{9\}$	$\{5\}$
3d	$\texttt{x} = \texttt{x}_{in}$	Input	$\texttt{x}_{in} = 5$	$\{2^{T2}\}$	$\{6\}$	$\{5, 12\}$	$\{2\}$

irrelevant calls to `f`). Each step (or row) represents an arc being traversed.

Step 0 represents the start of the traversal. In this step the slicing criterion is `print(a)` in thread `T1` (timestamp 16). The *Traversal State* is updated by annotating `T1` with the set of timestamps $\{16\}$. Step 1 represents the traversal of the interference arc from `print(a)` to `a = x` (in procedure `f`). This node has four timestamps that are updated in the state. Note that the traversal state is a countdown timer for each thread because the statements are traversed backwards. For this reason, `T1` had timestamp 16 and now it could be 6 or 13 (because there are two calls to `f` in `T1`). Step 2 traverses the flow arc back to the formal-in node $\texttt{x} = \texttt{x}_{in}$. After reaching the formal-in, the algorithm finds four possible input arcs to traverse, labelled with letters from (a) to (d). In the four cases, the timestamp of the node reached is smaller than the timestamp in the traversal state and there is an interprocedural threaded witness (in the tCFG) from the statement in *New Node* to the one in *Last Node* ($\texttt{x} = \texttt{x}_{in}$). Therefore, all four nodes could be executed before the last node (in different executions) and, thus, they are (wrongly) added to the slice, losing context-sensitivity: observe in Step 1 that traversing the interference arc leads us to procedure `f`. But, the algorithm travels from procedure `f` to all calls to `f` (in any thread) and not to the call from which we entered.

To solve the imprecision problems of the tSDG, we propose an alternative approach that takes advantage of the timestamp mechanism proposed by Nanda et al. to solve time travel; but restores the standard two-phase slicing algorithm proposed by Horwitz et al. We solve the interference dependence problem when transforming the set of tPDGs into the tSDG, thus being context-sensitive in the situations Nanda et al.'s tSDG was not.

6.3 The tSDG, revisited

In this section we propose a redefinition of the tSDG in such a way that: (i) it includes a new treatment for interference dependence so that the graph is context-sensitive in situations the old tSDG is not, (ii) it keeps the timestamp design to solve the time travel problem, and (iii) it can work with the standard two-phase algorithm, reducing complexity and improving performance.

Interference dependence (see Definition 6.1) connects definitions and uses that are performed concurrently. In some cases, the nodes connected may belong to the body of different procedures (see, e.g., the interference arc in Fig. 6.5). This means that interference can be an interprocedural data dependence, which breaks the principle of the SDG in which all data dependences enter to and exit from procedures throughout formal and actual nodes. This is the fundamental limitation of Nanda et al.'s tSDG. The key point of the new tSDG is to prevent interference arcs from freely jumping from the body of one procedure to the body of another one. That is, *interprocedural interference dependence* is not permitted.

One of the main problems of previous approaches is that formal and actual nodes, originally thought to represent dependences in the sequential execution of programs, are also used as source or target of interference arcs, which represent flow dependences of concurrent executions. This fact results in a mix of sequential and concurrent information managed by the same node, overloading it with conflicting responsibilities and leading to incomplete slices in some cases.

For this reason, the proposal of the new tSDG transfers this knowledge to a new group of nodes that only represent information about the concurrent execution: the formal and actual *interference nodes*. These nodes are analogous to the standard formal and actual nodes in sequential parameter passing, but they model concurrent parameter passing, i.e., they represent the value of a variable used/defined inside a procedure that has been defined/used in a thread executed in parallel:

- *Enter* nodes are augmented with new *formal-interference-in* (*formal-ifin*) and *formal-interference-out* (*formal-ifout*) nodes, control dependent on the *Enter* node. For each variable v that is a target of an interference dependence whose source is in another procedure, a formal-ifin node is generated with an assignment to copy the value of v (e.g., $v = v_{ifin}$). On the other hand, for each variable v that is the source of an interference dependence with target in another procedure, a formal-ifout node contains an assignment to copy any value of v (e.g., $v_{ifout} = v$) defined during the execution of the procedure.

- In a similar way, each procedure call is augmented with new *actual-interference-in* (*actual-ifin*) and *actual-interference-out* (*actual-ifout*) nodes, which are control dependent on the call node. At the call site, an actual-ifin node contains an assignment that copies the value of a shared variable v to a temporary variable (e.g., $v_{ifin} = v$). Complementarily, an actual-ifout node contains an assignment to copy the value of a shared variable v (e.g., $v = v_{ifout}$), defined inside the procedure.

In this model, as it happened with formal and actual nodes, formal and actual interference nodes associated to concurrent parameter passing are also connected by input arcs (from an actual-ifin to a formal-ifin) and output arcs (from a formal-ifout to an actual-ifout).

The addition of this new structure in procedure calls allows us to separate flow and interference dependences, connecting them to different actual nodes according to their sequential or concurrent nature. In our model, flow dependences are connected to actual-in/out nodes while interference dependences are connected to actual-interference-in/out nodes. We formalize these restrictions over flow and interference dependence as follows: (1) flow dependence ignores variable definitions and uses that occur in interference nodes, and (2) interference dependence ignores definitions and uses in formal/actual nodes, and is exclusively intraprocedural.

Definition 6.3 (tSDG-Flow Dependence)**.** *Let $G = (N, A)$ be a tSDG, and $n, m \in N$, two nodes. n is* tSDG-flow dependent *on m if:*

(i) *n is flow dependent on m (see Definition 2.8) and*
(ii) $n, m \notin (\text{FORMALIFNODES}(N) \cup \text{ACTUALIFNODES}(N))$.

Definition 6.4 (tSDG-Interference Dependence)**.** *Let $G = (N, A)$ be a tSDG, and $n, m \in N$, two nodes. n is* tSDG-interference dependent *on m if:*

(i) *n is interference dependent on m (see Definition 6.1),*
(ii) $\text{PROCEDURE}(m) = \text{PROCEDURE}(n)$, *and*
(iii) $n, m \notin (\text{ACTUALNODES}(N) \cup \text{FORMALNODES}(N))$.

6.3.1 Flow dependences generated by interference nodes

Although interference formal nodes are similar to formal nodes used in sequential parameter passing, the information represented by both is fundamentally different: while sequential formal nodes (formal-in) represent a definition of a variable v that always occur before executing the procedure's code or a use that always occur after executing the whole body of the procedure (formal-out), formal interference nodes represent definitions (formal-ifin) or uses (formal-ifout) of v that *may happen in any timestamp during the execution*

of the procedure's code. For this reason, in order to represent this specialized behaviour, we define a new dependence associated to formal interference nodes called *concurrent flow dependence*.

Definition 6.5 (Concurrent flow dependence)**.** *Let $G = (N, A)$ be a tSDG, and $n, m \in N$, two nodes. n is* concurrent flow dependent *on a preceding node m if:*

(i) m or n are formal interference nodes,
(ii) m defines a variable v,
(iii) n uses v, and
(iv) there exists a control-flow path from m to n.

This definition is really interesting because it allows v to be redefined in the control-flow path from m to n. Even in that case, n depends on m. The rationale comes from a fundamental property of interference formal nodes: they represent an interference that could happen at any instant during the execution of the procedure.

6.3.2 Timestamps in interference nodes

The addition of interference nodes forces us to take a small detour and ask ourselves the question: which timestamps should be assigned to interference nodes? Because we are using the timestamp-based method to solve time travel, all nodes must be labelled with timestamps. In order to establish a correct node chronology when numbering interference nodes, we need to consider all moments in which they could be defined or used. We start with formal interference nodes:

- Formal-interference-out nodes in a procedure represent all possible definitions of a shared variable. For this reason, the set of timestamps of these nodes includes the union of the timestamps of all the definitions of the variable in the procedure.
- Formal-interference-in nodes in a procedure represent a definition of a shared variable that occurred in a parallel thread and was used at some point in the procedure. Therefore, the set of timestamps associated to formal-ifin nodes is the union of all timestamps of uses of this variable in the procedure.

While formal interference nodes of a procedure contain the timestamps associated to *all* the calls to the procedure, actual interference nodes are only associated to those timestamps that happened during one specific call. They can be computed by selecting the timestamps that happened between the initial and final timestamps of the call.

6.3.3 An algorithm to generate the tSDG

Algorithm 6 describes how to transform a SDG into its associated tSDG. The SDG is formed from a set of nodes N and a set of control arcs (A_c), flow arcs (A_f), call arcs (A_{call}), input arcs (A_{in}), output arcs (A_{out}), and summary arcs (A_s) (see Section 6.1). The tSDG also includes the interference structure: formal-interference-in, formal-interference-out, actual-interference-in, and actual-interference-out nodes; and the arcs needed to connect them, including the interference arcs (A'_{if}) (induced by Definition 6.4) and the concurrent flow arcs (A'_{cf}) (induced by Definition 6.5).

There is a set of functions used in the algorithm that need to be explained beforehand. Given a node n, source or target of an interference, functions GETINTERFERENCEVAR(n) and GETTIMESTAMPS(n) return the name of the interfered variable in n and the timestamps associated to n, respectively. Function REPEATEDINTERFERENCE(n,l,v) is the termination condition of the recursion. It checks whether node n has been already processed with the same label l and the same interfered variable v. Function GETPROCEDUREROOT(n) returns the *Enter* node of the procedure where n is contained. Function ADDNODE(N',l,t) creates (and returns) a new node in N' with label l and timestamp t. If the node already exists, t is added to the already existing set of timestamps. Function GETCALLS(n) returns a set with all the call nodes that call the procedure whose *Enter* node is n. Finally, given a call node c and a set of timestamps t, function FILTERTIMESTAMPS(c,t) returns the timestamps that occur inside c.

The algorithm starts by procesing all interprocedural interference dependences. The structure that represents each interprocedural interference dependence is created by means of functions ADDINTERFERENCEINNODES and ADDINTERFERENCEOUTNODES. While the call to ADDINTERFERENCEINNODES (line 3) creates formal-interference-in and actual-interference-in nodes, the call to ADDINTERFERENCEOUTNODES (line 4) creates formal-interference-out and actual-interference-out nodes.

In each call to functions ADDINTERFERENCEINNODES and ADDINTERFERENCEOUTNODES, we start by extracting the variable involved in the interference and checking whether the input node n has already been processed. In that case, the function returns and nothing is generated (lines 11 and 25). Otherwise, we extract the root (*Enter*) node of the procedure and the set of timestamps associated to n. Then, according to whether it is an input or output node, a different formal interference node is created (*newFormal*) and connected to the *Enter* node with a control arc (lines 14–15 and 28–29), and the corresponding concurrent flow arc adds the connection between the formal node and n (lines 16 and 30).

Algorithm 6 tSDG interference structure creation algorithm

Input: A SDG $G = (N, A_c \cup A_f \cup A_{call} \cup A_{in} \cup A_{out} \cup A_s)$
Output: A tSDG $G' = (N', A'_c \cup A_f \cup A_{call} \cup A'_{in} \cup A'_{out} \cup A'_s \cup A'_{if} \cup A'_{cf})$
Initialization: $N' = N$, $A'_c = A_c$, $A'_{in} = A_{in}$, $A'_{out} = A_{out}$, $A'_{if} = \emptyset$, $A'_{cf} = \emptyset$,
I_{inter} contains all interprocedural interference dependences,
I_{intra} contains all intraprocedural interference dependences

1: **begin**
2: **for** $(n, n') \in I_{inter}$ **do**
3: AddInterferenceInNodes(n')
4: AddInterferenceOutNodes(n)
5: **for** $(n, n') \in I_{intra}$ **do**
6: $A'_{if} = A'_{if} \cup \{(n, n')\}$
7: $A'_s \leftarrow$ ComputeSummaryArcs(G)
8: **end**
9: **function** AddInterferenceInNodes(n)
10: $v \leftarrow$ GetInterferenceVar(n)
11: **if** RepeatedInterference(n, "in", v) **then return**
12: $root \leftarrow$ GetProcedureRoot(n)
13: $tstamps \leftarrow$ GetTimestamps(n)
14: $newFormal \leftarrow$ AddNode(N', "$\mathtt{v} = \mathtt{v}_{ifin}$", $tstamps$)
15: $A'_c \leftarrow A'_c \cup \{(root, newFormal)\}$
16: $A'_{cf} \leftarrow A'_{cf} \cup \{(newFormal, n)\}$
17: **for** $c \in$ GetCalls($root$) **do**
18: $callTstamps \leftarrow$ FilterTimestamps(c, $tstamps$)
19: $newActual \leftarrow$ AddNode(N', "$\mathtt{v}_{ifin} = \mathtt{v}$", $callTstamps$)
20: $A'_c \leftarrow A'_c \cup \{(c, newActual)\}$
21: $A'_{in} \leftarrow A'_{in} \cup \{(newActual, newFormal)\}$
22: AddInterferenceInNodes($newActual$)
23: **function** AddInterferenceOutNodes(n)
24: $v \leftarrow$ GetInterferenceVar(n)
25: **if** RepeatedInterference(n, "out", v) **then return**
26: $root \leftarrow$ GetProcedureRoot(n)
27: $tstamps \leftarrow$ GetTimestamps(n)
28: $newFormal \leftarrow$ AddNode(N', "$\mathtt{v}_{ifout} = \mathtt{v}$", $tstamps$)
29: $A'_c \leftarrow A'_c \cup \{(root, newFormal)\}$
30: $A'_{cf} \leftarrow A'_{cf} \cup \{(n, newFormal)\}$
31: **for** $c \in$ GetCalls($root$) **do**
32: $callTstamps \leftarrow$ FilterTimestamps(c, $tstamps$)
33: $newActual \leftarrow$ AddNode(N', "$\mathtt{v} = \mathtt{v}_{ifout}$", $callTstamps$)
34: $A'_c \leftarrow A'_c \cup \{(c, newActual)\}$
35: $A'_{out} \leftarrow A'_{out} \cup \{(newFormal, newActual)\}$
36: AddInterferenceOutNodes($newActual$)

The next step is to extract all the call nodes to generate the corresponding actual nodes. For each call c, we create and add to the graph the corresponding actual interference node ($newActual$) and connect it to c with a control arc (lines 19–20 and 33–34). Then, actual and formal interference nodes are connected with the corresponding input/output arcs (lines 21 and 35).

After processing all interprocedural interference dependences, we convert all intraprocedural interference dependences into interference arcs (A'_{if}) (lines 5-6).

Finally, we can generate the summary arcs with the standard algorithm (line 7). This completes the final tSDG, which includes the new concurrent parameter passing structure. xr

Example 6.6 (Transforming the SDG to a tSDG)**.**
Consider the code in Fig. 6.6, with two threads `T1` and `T2` that read and write over the shared variable x. The same figure shows how the SDG of the program (left) is transformed into the tSDG (right). According to Definition 6.1, this program contains four interprocedural interference dependences (not drawn because the SDG does not include interferences): two from `x = 2` in `T1` to `a = x` and `x = x + 1` in `T2`, and other two from `x = x + 1` and `x = a` in `T2` to `print(x)` in `T1`. These four dependences stop being interprocedural in the tSDG: they only enter and exit procedure `f` through the corresponding interference nodes (those in grey).

When applying the transformation from SDG to tSDG, the four interprocedural interferences are processed by Algorithm 6 producing: (i) the set of interference nodes (grey nodes), (ii) the input and output arcs connecting them, (iii) the set of concurrent flow arcs associated to procedure `f` (light blue arcs), and (iv) the set of interference arcs. As statements 13 and 14 in procedure `f` are target of an interprocedural interference dependence (they both use shared variable `x`), a formal-ifin assignment for `x` is created, containing all their associated timestamps (6 and 7). On the other hand, since statements 14 and 15 are source of interprocedural interference dependences (they define shared variable `x`), a formal-ifout assignment for `x` is also included in the tSDG with the timestamp of both definition nodes (7 and 8). The formal-interference structure is replicated in the corresponding call to procedure `f`, generating actual interference nodes with the same timestamps (they are associated to the only call to procedure `f`). Finally, following Definition 6.5, concurrent flow arcs are generated in `f` from formal-ifin nodes (respectively to formal-ifout nodes).

It is interesting to remark that flow arcs only connect actual and formal nodes but not interference nodes (according to Definition 6.3), while concurrent flow arcs only connect interference nodes (according to Definition 6.5),

```
1  procedure main()
2    x = 1;
3    cobegin{
4      // Thread T1
5      x = 2;
6      print(x);
7    }{
8      // Thread T2
9      call f;
10   }
11
12 procedure f
13   a = x;
14   x = x + 1;
15   x = a;
```

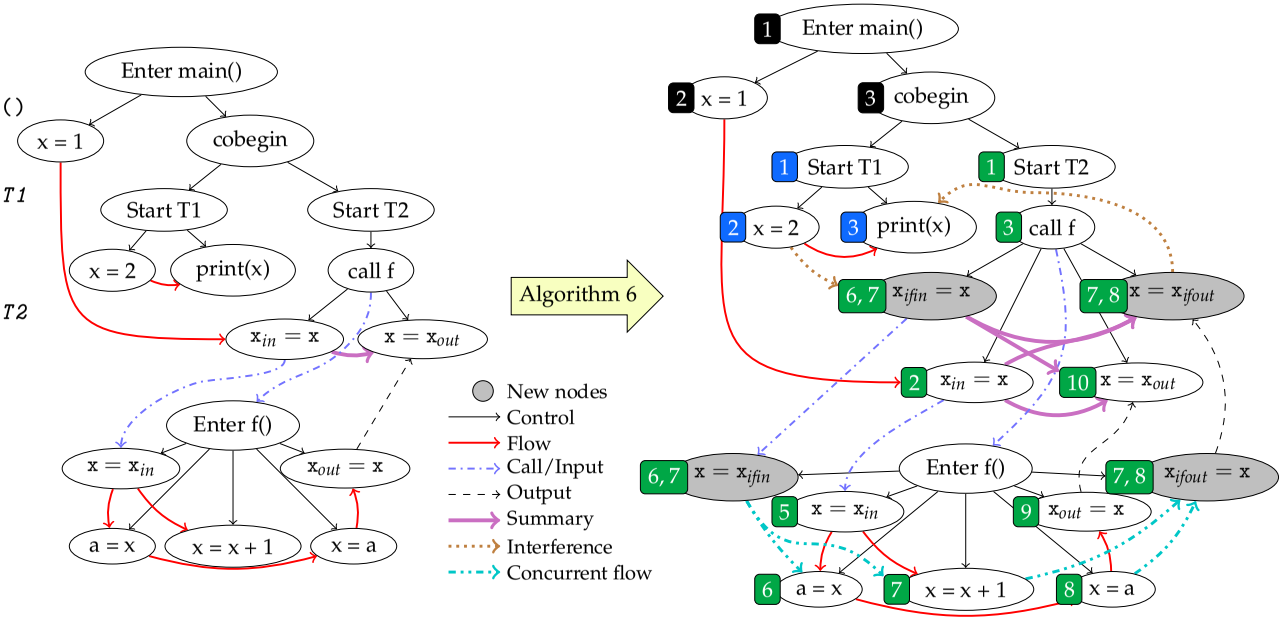


FIGURE 6.6: Transformation from SDG to tSDG in a program with two threads and a procedure call.

dividing responsibilities.

It is worth mentioning that our tSDG generates new summary arcs (see Fig. 6.6) that account for all execution paths induced by concurrent scenarios. This is the key point why our model solves the incompleteness problem (see Section 6.2 and cf. Nanda et al. [79, Section 3.1]). Instead of replicating the behaviour of summary arcs with interprocedural interference arcs and ad-hoc changes to the slicing algorithm, our solution naturally uses the same summary mechanisms and slicing algorithm as the standard SDG.

Finally, we state an important property of the tSDG: all interferences are represented with a chain of dependences. This is especially relevant because the tSDG does not contain interprocedural dependence arcs, but thanks to that property they are represented through other dependence chains.

6.4 Slicing the tSDG

The slicing algorithm used to slice our tSDG model is the standard two-phase SDG slicing algorithm, but including the timestamp mechanism proposed by Nanda et al. to avoid time travel. The new nodes (interference nodes) introduced in the tSDG have been conveniently labelled with timestamps so that they can be directly processed with Nanda et al.'s timestamp algorithm.

Example 6.7 (Slicing our tSDG)**.**
Consider again the code in Fig. 6.1 and the slicing criterion $\langle 7, a \rangle$. The new tSDG associated to this program is shown in Fig. 6.7. We can compare it with the old tSDG shown in Fig. 6.5. First, the slice computed with the new tSDG is the minimal slice shown in Fig. 6.1a. The (imprecise) slice computed with the old tSDG is shown in Fig. 6.1b.

There are two differences that can be clearly observed: (i) the new tSDG does not allow interprocedural interference arcs. In contrast, the old tSDG contains an interprocedural interference arc (it connects a statement inside `f`'s body with a statement outside `f`). Precisely for this reason, (ii) in the old tSDG some nodes in procedure `f` are reached during slicing Phase 1 while in the new tSDG all the nodes in `f` can only be reached during slicing Phase 2. These two facts are the key factors for obtaining the expected slice with the tSDG.

Table 6.3 shows how the slicing algorithm prevents the traversal of input arcs (because they are reached in Phase 2) and, thus, excludes the calls to `f` in lines 3 and 5. The traversal is divided into five main blocks (separated by horizontal lines): the first block (steps 1a and 2aa) represents the path in Phase 1 from the slicing criterion to $\mathtt{x}_{in} = 4$. The second block (steps 1b and

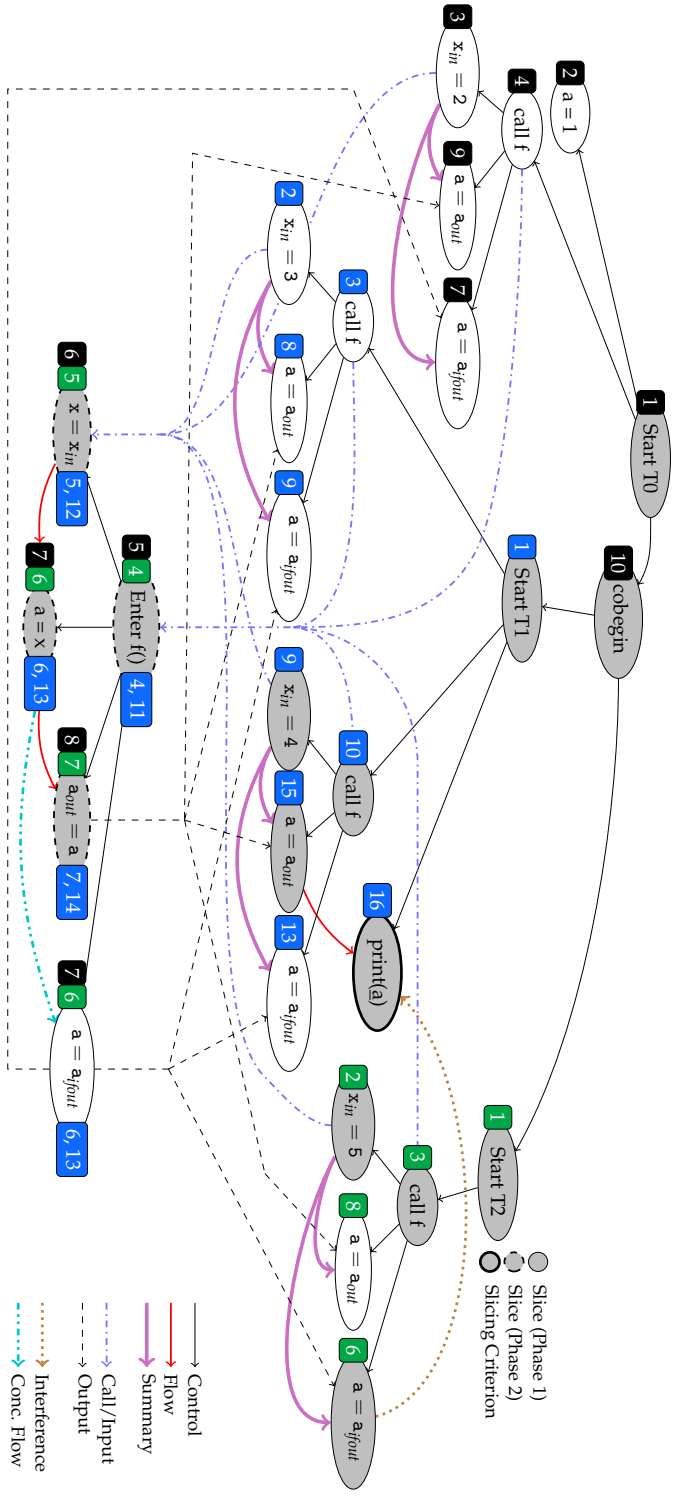


FIGURE 6.7: New tSDG of the program in Fig. 6.1, sliced w.r.t. $\langle 7, a \rangle$.

TABLE 6.3: New tSDG slicing traversal of the program in Fig. 6.1 (from the slicing criterion $\langle 7, a\rangle$ to all the actual-in nodes $x_{in} = x$ in the calls to procedure f).

Step	Last Node	Arc Type	Phase	New Node Statement	New Node Timestamps	Traversal State T0	Traversal State T1	Traversal State T2
0	-	-	1	print(a)	$\{16^{T1}\}$	$\bot$	{16}	$\bot$
1a	print(a)	Flow	1	$a = a_{out}$	$\{15^{T1}\}$	$\bot$	{15}	$\bot$
2aa	$a = a_{out}$	Summ.	1	$x_{in} = 4$	$\{9^{T1}\}$	$\bot$	{9}	$\bot$
1b	print(a)	Interf.	1	$a = a_{ifout}$	$\{6^{T2}\}$	$\bot$	{16}	{6}
2ba	$a = a_{ifout}$	Summ.	1	$x_{in} = 5$	$\{2^{T2}\}$	$\bot$	{9}	{2}
2ab	$a = a_{out}$	Output	2	$a_{out} = a$	$\{8^{T0}, 7^{T1}, 14^{T1}, 7^{T2}\}$	{8}	{7,14}	{7}
3a	$a_{out} = a$	Flow	2	a = x	$\{7^{T0}, 6^{T1}, 13^{T1}, 6^{T2}\}$	{7}	{6,13}	{6}
2bb	$a = a_{ifout}$	Output	2	$a_{ifout} = a$	$\{7^{T0}, 6^{T1}, 13^{T1}, 6^{T2}\}$	{7}	{6,13}	{6}
3b	$a_{ifout} = a$	C. Flow	2	a = x	$\{7^{T0}, 6^{T1}, 13^{T1}, 6^{T2}\}$	{7}	{6,13}	{6}
4	a = x	Flow	2	$x = x_{in}$	$\{6^{T0}, 5^{T1}, 12^{T1}, 5^{T2}\}$	{6}	{5,12}	{5}

2ba) represents the path in Phase 1 from the slicing criterion to $x_{in} = 5$. The rest of nodes collected in Phase 1 are not represented in the table because they are trivially reached through control arcs. In Phase 2, the third (steps 2ab and 3a) and fourth (steps 2bb and 3b) blocks respectively represent the paths from $a = a_{out}$ and $a = a_{ifout}$ to `a = x`. Finally, the last block traverses a flow arc to reach the formal-in node $x = x_{in}$. Since we are in Phase 2, the four input arcs cannot be traversed. This avoids to unnecessarily collect nodes $x_{in} = 2$, $x_{in} = 3$, and, thus, the calls to `f` in lines 3 and 5. Note, however, that nodes $x_{in} = 4$ and $x_{in} = 5$ where already collected by summary arcs in Phase 1 during steps 2aa and 2ba, keeping the slice complete, and the slicing traversal context-sensitive.

Complexity

In this section we compute the asymptotic size of the tSDG, the temporal cost of building it, and the complexity of the slicing algorithm. The size of the tSDG can be measured in terms of the following parameters:

- S_{if}: statements that are source or target of an interprocedural interference dependence.
- V_i: number of different variables causing interprocedural interference dependences at statement i.

- C_i: number of calls to procedures that may be in execution while statement i is being executed, i.e., procedures that may directly or transitively execute i.
- P_i: number of procedures that contain at least one of the calls in C_i.
- F_i: number of procedures in P_i not called by any other procedure.

The tSDG increments the number of nodes of the SDG by $\sum_{i \in S_{if}} (C_i + P_i)$. This formula can be easily obtained from Algorithm 6. In the tSDG, P_i represents formal-ifin/ifout nodes and C_i represents actual-ifin/ifout nodes. On the other hand, the tSDG increments the number of arcs in:

$$\sum_{i \in S_{if}} V_i + 2(C_i + P_i) - F_i$$

because for each statement source or target of interprocedural interference dependence, V_i interference-flow arcs are generated for each statement in S_{if}. Then, we generate two more arcs (one control arc and one interprocedural arc) for all generated formal and actual nodes except for those in F_i (for them, we only generate one control arc).

Constructing a tSDG $G = (N, A)$ has a quadratic cost $\mathcal{O}(N^2)$, exactly the same as the SDG (and the old tSDG). This is due to the fact that the most expensive task is computing summary arcs (a quadratic operation) that is also done in the standard SDG.

Finally, the complexity of the slicing algorithm is the same as in the old tSDG. The timestamp mechanisms used to compute and validate threaded witnesses makes the complexity of the algorithm $\mathcal{O}(N^{d^t})$, where d is the max depth of the call graph[1] and t the number of threads [79].

While keeping the same asymptotic cost, our algorithm is more efficient than Nanda et al.'s algorithm thanks to the improved treatment of context-sensitivity. Since our slicing algorithm uses the two-phase traversal proposed by Horwitz et al., the maximum number of nodes handled by the algorithm with the same traversal state is N (while in Nanda et al.'s approach is $2N$), being the runtime complexity for managing timestamps at each node $\mathcal{O}(N^3)$ and traversing each node to manage context-sensitivity in our algorithm $\mathcal{O}(N + A)$ (every node and arc is only processed once with the same traversal state).

The termination of the slicing phase is ensured by the fact that whenever a node is visited with the same timestamps, the traversal finishes. Therefore, since the number of nodes to visit is finite and each node is visited at most a number of finite times, termination is ensured. In particular, a node can

[1]The maximum number of arcs in an acyclic path [25].

only be visited as many times as combinations of timestamps exists, and the number of timestamps is also finite.

6.5 Related work

The first concurrency-aware graph representation model was proposed by Cheng [14, 15]. He defined a graph-like representation model called *Process Dependence Net* (PDN), which represented five types of primary program dependences in concurrent ADA programs. Cheng was the first author that reduced the problem of slicing concurrent programs to a graph reachability problem. Zhao [109] introduced a *multi-thread dependence graph* to represent concurrent Java programs. His approach replicated the classic two-phase slicing algorithm for sequential interprocedural programs, traversing interference and synchronization dependences transitively. Something similar happened with the approach proposed by Hatcliff [48]. Hatcliff specialized extra dependences in concurrent Java programs while still treating interference dependences as transitive. Because they modelled interference dependence as transitive, all three approaches did not solve the time travel problem, producing inaccurate slices.

There are several works that have tried to solve the interference intransitivity for generating accurate slices and work around the time travel problem. Zhang et al. [107] calculated a predecessor set, which ensures that every node in the slice may be executed before the slicing criterion, since all the statements included in a backward slice must be computed before it. On the other hand, Chen et al. [110] discarded the two-phase algorithm and computed what they called dependence sequences. They used these sequences to compute slices as the difference between the sequences reached from the slicing criterion and the sequences formed by nodes that actually cannot be reached due to time travel. Although these two approaches proposed novel ideas, they were not able to completely resolve time travel because both algorithms could not ensure that all nodes in every dependence sequence formed a valid path. The above approaches were the first proposals to deal with concurrent dependences and with the time travel problem.

In 1998, Krinke [60] designed the *threaded Program Dependence Graph* (tPDG), an intraprocedural model based on the PDG that introduced the definition of interference dependence based on threaded witnesses. Krinke's approach was the first to define a precise slicing algorithm for programs with shared memory that completely solved the time travel problem. However, the summary arcs and the two-phase algorithm introduced by the SDG were not compatible with Krinke's approach, generating context-insensitive slices and

inaccuracies in the presence of nested threads and threads nested within loops. Five years later, the same Krinke proposed in [59] a context-sensitive version of his approach for program slicing. His approach defined a new program representation model called the *threaded Interprocedural Program Dependence Graph* (tIPDG), built upon the *Interprocedural Threaded Control-Flow Graph* (ITCFG). The tIPDG was the first context-sensitive approach to slice concurrent programs. In order to obtain accurate context-sensitive slices, the tIPDG labels interprocedural arcs to avoid visiting an actual-in node of a call if its corresponding actual-out node has not been previously included in the slice. Although this approach was context-sensitive, it was much slower than either a context-insensitive concurrent slicer or the SDG. The main difference between Krinke's approach and ours is the method used during the traversal to reach context-sensitivity. Krinke implements a single-phase algorithm that does not require summary arcs and collects labels during the traversal of interprocedural arcs, reaching context-sensitivity with an exponential cost. In contrast, our model uses the linear-cost two-phase slicing algorithm proposed by Horwitz et al., which manages traversal restrictions according to two differentiated slicing phases. In our approach, summary arcs computed during the tSDG construction process are essential to ensure completeness because they represent the information flow of interprocedural interference dependences, which are explicitly represented in Krinke's tIPDG.

Nanda et al. [80] adapted Krinke's traversal algorithm and refined it, defining flow dependences induced by threads generated within loops and nested threads, and providing an alternative way of computing threaded witnesses. Their slicing algorithm was more efficient and remained precise in the presence of the mentioned thread combinations. In a later publication [79] they introduced a more efficient quasi-context-sensitive approach. This approach solved some completeness and correctness issues of their previous work, but left room for improvement in its context-sensitivity as shown in Section 6.2. We use the same system to solve the time travel problem as Nanda et al. The main difference between their approach and ours are (i) the way interferences arcs are computed and (ii) the modifications in the slicing algorithm required to ensure completeness. While they allow interprocedural interference arcs, we force all interference arcs to be intraprocedural and we create new nodes to propagate interferences from one procedure ot another through input and output arcs. This fact allows us to use the standard Horwitz et al.'s slicing algorithm, while the old tSDG needs to perform phase changes during the traversal to compute complete slices.

Later approaches have inherited either Krinke's or Nanda et al.'s approaches to context-sensitivity. A novel approach to computing threaded

witnesses is provided in [89], where the authors produced a graph that combines the tSDG and the possible states of the traversal proposed by Nanda et al., effectively encoding threaded witnesses into the graph and simplifying the traversal algorithm. This proposal is orthogonal to our technique. Therefore, it could be also applied to our graph, speeding up the time required to slice our model.

Chapter 7

Undecidability of data dependence

> I don't know half of you half as well as I should like; and I like less than half of you half as well as you deserve.
>
> *J. R. R. Tolkien, The Fellowship of the Ring*

The word *undecidability* is often related (in computer science) to the decision problem. A problem that poses a question with YES-NO answers is called undecidable if there does not exist an algorithm that terminates with the correct answer for every instance of the problem. The Hilbert's Entscheidungsproblem [50], the halting problem [19], and Rice's theorem are examples of this notion of undecidability.

This chapter shows that it is impossible to construct an algorithm that captures all, and only, data dependences in any program with complex data structures or data types with an infinite domain. More precisely, we cannot construct an algorithm that can answer in finite time the following question for any program, statement, and variable: Can statement s influence the value computed at this variable v? (does v depend on s?).

One of the most important results in the computability theory was stated by Henry Gordon Rice in 1951 [95]. He stated a fundamental result (today known as Rice's theorem) about the undecidability of properties:

"*All non-trivial semantic properties of programs are undecidable.*"

It is important to remark that, in Rice's theorem, a property p of a program is defined as a set of recursively enumerable languages, and a property is non-trivial if there exists a recursively enumerable language having the property p, and there exists a recursively enumerable language not having the property p (i.e., p is not empty nor it includes all recursive enumerable languages).

The implications of this result are of paramount importance, and it imposes limits to the computation theory. Moreover, this result is directly applicable to the field of partial functions. According to Rice's theorem, given a non-trivial property p, we can always find a program for which the property is undecidable. Of course, this does not mean that we cannot find a set of programs for which the property is decidable. For instance, non-termination is undecidable in general, but it is decidable for programs that do not have loops.

Therefore, Rice's theorem is too general because it considers the class that contains all programs, thus it can be complemented with theoretical results that restrict the set of programs over which we want to demonstrate a particular property (or equivalently, analysis). This is what was done by Landi [65], and later Ramalingan [90], who advanced the field by proving that *alias analysis* is undecidable for general purpose computer languages. The same was done later by Reps [93], who proved that context-sensitive field-sensitive data dependence analysis (DDA) is undecidable for programs with functions and composite data structures. We already know thanks to Rice's theorem that any non-trivial static analysis is undecidable, being one fundamental reason non-reachability. Clearly, any question about a point in a program is undecidable because that point could not be reached due to non-termination. Reps formulated an interesting (and fundamental) research question: Is non-reachability the (only) fundamental cause of undecidability? Reps proved that, for context-sensitive field-sensitive DDA, the answer is "no". He proved that this analysis is undecidable even if we assume that the point of interest is always reachable (i.e., if we ignore non-termination by assuming that all paths in the control flow graph are computable). It is important to remark the importance of this result because it subsumes many other previous results: because Reps found a problem that cannot be solved with a context-sensitive analysis, then it can neither be solved with a context-insensitive (less powerful) analysis. From this fact, we can derive that context-insensitive DDA and field-insensitive DDA are also undecidable.

Figure 7.1 shows different classes of programs. Class A includes all programs. According to Rice's theorem, DDA is undecidable for this set. It is also well-known that DDA is undecidable for non-terminating programs (class B). Reps proved that DDA is undecidable for class C. Specifically, he found a program in class C for which DDA is undecidable. The undecidability of DDA for the classes D, E, and F remains unknown.

This is precisely the research question of this work: Is DDA decidable for all programs in classes D, E, and/or F? or, conversely, can we find a program in sets D, E, and/or F for which DDA is undecidable?

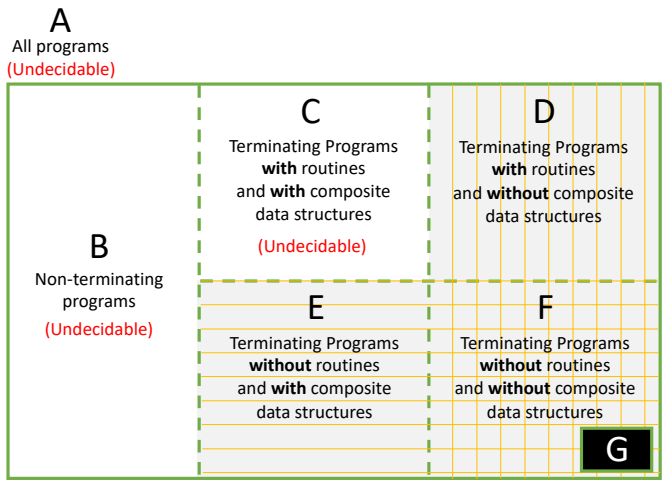


FIGURE 7.1: Undecidability of data dependence analyses for different program classes

In this work, we expand Reps' results to answer the above questions, proving that DDA is also undecidable for E. We also generalize our results for other analyses such as data-flow analysis (DFA). We go further and identify specific requirements of a programming language to make DDA (and other analyses) undecidable. I.e., we characterize a set of programs that induce a non-recursively enumerable language. The complement of this set of programs supposedly induces a recursively enumerable language with respect to the given analyses, but we haven't proved it. This question remains open. In Turing machine terms, this means that for languages that do not satisfy the requirements that we state and belong to C and E; there supposedly (we have not proved it) exists a Turing machine that halts on all inputs and accepts the identified language; and the converse, for languages that satisfy the requirements, there definitely (we have proved it) does not exist such a Turing machine.

This chapter is structured as follows. First, in Section 7.1 we recall three different program analyses: the data dependence analysis, the data-flow analysis, and the Reps' data dependence analysis (RDA). In Section 7.2, we present a variant of the *post's correspondence problem* [86], which is a problem that has been proved undecidable. Next, in Section 7.3 we showcase two programs in classes C, and E; for which solving a RDA implies solving the post's correspondence problem; and, thus, for these classes of program's RDA is undecidable. We also extend our results on RDA to DFA and DDA.

7.1 Data Dependence and Data-Flow Analyses

Data dependence analysis (DDA) [73, 82] is an area of compiler theory which tries to identify data dependences in a program. Roughly, a statement s_2 is data dependent (also known as flow dependent) on another statement s_1 if the values computed at s_2 can be influenced (in some execution) by the values computed at s_1 (i.e., in at least one execution, if we modify s_1 then the values computed at s_2 change with respect to those computed if s_1 is not modified).

Data-flow analysis (DFA) [2, 45] is another area which tries to compute the set of values calculated at a given point in the program. The most representative example is *reaching definitions*, which statically determines what variable definitions[1] may reach a given point in the code. Formally,

Definition 7.1 (Reaching Definitions)**.** *Let s_1 and s_2 be two statements in a program P, where s_1 defines variable x. We say that the definition of variable x at s_1* reaches *s_2 if there exists a realizable control-flow path from s_1 to s_2 where x is not redefined (i.e., it is not killed in all paths between s_1 and s_2).*

Data dependence is a more sophisticated analysis that uses the reaching definitions analysis. It is very similar to Definition 2.8, but includes transitivity in its definition (instead of relying on the slicing algorithm to make it transitive). Formally,

Definition 7.2 (Data Dependence)**.** *Let s_1 and s_2 be two statements in a program P. s_2 is* data dependent *on s_1 if*

- *Direct dependence: (i) s_1 defines a variable x, (ii) s_2 uses x, and (iii) the definition in s_1 reaches s_2; or*
- *Transitive dependence: there exists a statement s_3 such that s_2 is data dependent on s_3 and s_3 is data dependent on s_1.*

Example 7.1 (Data-related analyses)**.**
The following pair of simple programs illustrate three different analyses:

```
1       x = [];                 x = [];
2       x = [x];                x = x;
3       y = x;                  y = x;
```

Q1 Could the definition of x in line 1 reach line 3?
Q2 Could variable x in line 1 $(1, x)$ influence $(3, y)$?
Q3 Could the value of y in line 3 be the same value of x in line 1?

[1] Along the chapter, we adopt the data-flow analysis terminology's term *define* (used, e.g., in the *def-use* chains). This term should be understood as assign/kill (in compiler terminology).

Q1 can be answered by a DFA: the answer is *no* for both programs because the definition of x in line 1 is killed by the definition of x in line 2. Q2 can be answered by a DDA: the answer is *yes* for both programs because $(3, y)$ depends on $(2, x)$, which in turn depends on $(1, x)$. Q3 needs a different analysis. Note that, to answer Q3, we require an analysis that is more restrictive than a DDA because the value of $(3, y)$ must be the same as the value of $(1, x)$; but we need an analysis less restrictive than a DFA because it is allowed that the definition in line 1 is killed. The answer is *no* for the program on the left and *yes* for the program on the right. We call this third analysis "Reps' data analysis" (RDA), because it was used by Thomas Reps in [93].

RDA is a especial kind of DDA that can be defined as follows:

Definition 7.3 (Reps' Data Dependence (RDA))**.** *Let s_1 and s_2 be two statements in a program P. Variable y at s_2 is* Reps' data dependent *on variable x at s_1 if*

- *Direct propagation: variable x at s_1 reaches s_2 and s_2 assigns x to y, or*
- *Transitive propagation: there exists a statement s_3 defining a variable z such that variable y at s_2 is* Reps' data dependent *on variable z at s_3 and variable z at s_3 is* Reps' data dependent *on variable x at s_1.*

Determining data dependences is useful in a number of disciplines such as program analysis [77], automatic parallelization [5], dead code elimination [96], and program slicing [99], among many others. In the general case (for class A), all the analyses mentioned above are undecidable, being the reason non-termination. For an arbitrary program, it is not possible to determine if a statement will ever be executed (the program could diverge before reaching that point); therefore, we cannot decide if the statement is dependent (in any way) on another statement. The interesting questions is therefore, the following: If we know that the statements of interest will be reached (for instance, in debugging, one can see that a variable has actually been reached producing a wrong value, and we want to know what statements affected that variable), is my analysis decidable?

Landi [65], later Ramalingan [90], and Reps [93] answered this question for different analyses assuming that all execution paths are realizable (ignoring non-termination). Under this condition, although data dependence analysis is well-known as a difficult problem, it has remained unclear for which classes of programs it is a decidable problem, if ever. There are, however, important results about specific subproblems. For instance, Thomas Reps proved that context-sensitive field-sensitive[2] RDA is undecidable for class C (programs

[2]In his paper, Reps used the term *structure-transmitted*; but nowadays, the term *field-sensitive* is more extended.

with functions and composite data structures) [93]. In this chapter, we extend Reps' results in two complementary directions: (i) we prove that RDA is undecidable for programs without functions (class E), and (ii) we also prove that the above results are also applicable to DDA and DFA, thus we also prove that DFA and DDA are undecidable for classes C and E. More specifically, they are undecidable for the class of programs that satisfy three conditions:

1. the program has a conditional jump statement (e.g., `if`),
2. the program uses recursion or iteration (e.g., `while`), and
3. the program has complex data types (arrays, tuples, objects) or an infinite data type (e.g., strings, Python integers).

We show that these conditions are enough to construct a program for which any DFA/DDA/RDA is undecidable. Unfortunately, the class of programs that satisfy these conditions is extremely wide. Therefore, we conclude that for non-trivial programs, DFA/DDA/RDA are undecidable *even if all execution paths are realizable*.

7.2 The Parenthesis-Post Correspondence Problem

Our undecidability results are based on a variant of an already known undecidable problem (the parenthesis-post correspondence problem (P-PCP) [93]) that we show equivalent to our problem. In particular, we show that deciding DFA/DDA/RDA for some specific classes of programs implies deciding a P-PCP.

Definition 7.4 (Parenthesis-Post Correspondence Problem)**.** *An instance of* Parenthesis-Post's Correspondence Problem*, or P-PCP, consists of two lists of strings, X and Y, each containing k strings: $X = x_1, x_2, \ldots, x_k$; $Y = y_1, y_2, \ldots, y_k$; with $\forall x \in X \,.\, x \in \{(, [\}^+$ and $\forall y \in Y \,.\, y \in \{),]\}^+$. A solution to an instance of P-PCP is defined with the aid of the following linear context-free grammar:*

$$\begin{aligned} \mathit{balanced} \rightarrow\ & (\ \mathit{balanced}\ ) \\ \mid\ & [\ \mathit{balanced}\ ] \\ \mid\ & \epsilon \end{aligned}$$

An instance of P-PCP has a solution if there exists a non-empty sequence of integers $S = i_1, i_2, \ldots, i_n$ $(n > 0)$ such that:

1. $\forall i \in S \,.\, 1 \leq i \leq k$*, and*
2. $x_{i_1} x_{i_2} \ldots x_{i_n} y_{i_n} \ldots y_{i_2} y_{i_1} \in L(\mathit{balanced})$.

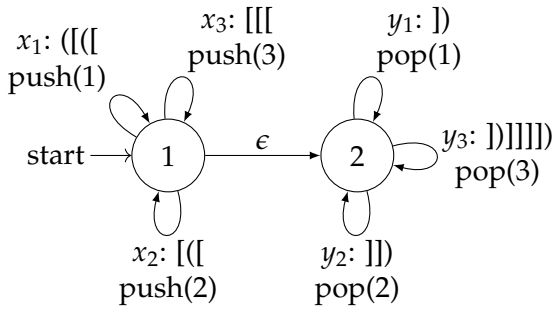


FIGURE 7.2: A pushdown automaton that represents the P-PCP instance from Example 7.2. When the traversal stops, the sequence pushed to the stack in state 1 is a valid solution if (i) the final stack is empty, and (ii) the word obtained belongs to $L(balanced)$.

Example 7.2 (Solving an instance of P-PCP)**.**
Consider the following instance of P-PCP (based on Example 2.2 from [93]):

$$\begin{array}{lll} X = & ([([, & [([, \quad [[[\\ Y = &]), &]]), \quad])]]]]) \end{array}$$

One of its solutions is $S = 1, 2, 3, 1$ because it fulfils the conditions:

$$\begin{array}{cccccccccc} x_1 & x_2 & x_3 & x_1 & y_1 & y_3 & y_2 & y_1 & = \\ ([([& [([& [[[& ([([&]) &])]]]]) &]]) &]) & \in L(balanced) \end{array}$$

However, finding a solution to this P-PCP instance is not trivial. The PDA in Fig. 7.2 shows the possible (infinite) paths that a search must traverse to find a solution.

The undecidability of P-PCP is stated in [93]. It is a variant of the Post's Correspondence Problem (PCP), which had been proved undecidable (cf. Hopcroft and Ullman [14, pp. 193-198], Lewis and Papadimitriou [23, pp. 289-294], and Harrison [8, pp. 249-256]).

7.3 Undecidability of Data Analyses

In this section, we prove the undecidability of Reps' data analysis (RDA) for two classes of programs. We include an example similar to Reps' RDA for programs with functions and with composite data structures (F^+C^+RDA) and our own example for programs without functions (F^-C^+RDA). Both are

```
1  a = 42 # A unique value that does not appear again
2  def f1():
3    a = [0, [[0, [a, 0]], 0]]  # x1: ([([
4    if cond1: f()
5    a = a[1][0]                # y1: ])
6
7  def f2():
8    a = [0, [[0, a], 0]]       # x2: [([
9    if cond2: f()
10   a = a[1][1][0]             # y2: ]])
11
12 def f3():
13   a = [0, [0, [0, a]]]       # x3: [[[
14   if cond3: f()
15   a = a[1][0][1][1][1][1][0] # y3: ])]]]])
16
17 def f():
18   if   cond4: f1()
19   elif cond5: f2()
20   else:       f3()
21
22 f()
23 # Q1: Could a be 42 here?
```

FIGURE 7.3: A Python program whose F^+C^+RDA is equivalent to solving the P-PCP instance from Example 7.2.

written in Python, but each can be adapted to most programming languages, if they have (i) potentially infinite loops or recursion, and (ii) data types with an infinite set of possible values. Then, we extrapolate these results to data-dependence analyses (DDA) and even data-flow analyses (DFA).

7.3.1 RDA with functions and composite data structures (F^+C^+RDA)

In January 2000, Thomas Reps [93] proved the undecidability of context-sensitive field-sensitive RDA for class C by showing that any instance of P-PCP can be codified in a program with functions and with composite data structures. He showed that the data structures of a program can encode the two lists of strings of P-PCP (they are an instance of P-PCP), and finding the path that solves the data dependence question would be equivalent to solving the P-PCP instance. Example 7.3 shows a program equivalent to the one presented by Reps.

Example 7.3 (Counterexample for F$^+$C$^+$RDA undecidability)**.**
Consider the program in Fig. 7.3, where three functions (`f1()`, `f2()`, and `f3()`) are called an arbitrary number of times and in an arbitrary order because all calls are inside undefined conditional expressions (see `condX` in lines 4, 9, 14, 18, 19, 20). For the same reason, this sequence of nested calls can end at any call (whenever a conditional in lines 4, 9, or 14 is evaluated to false).

Every time we enter into a function, the value of `a` (initially 42) is encapsulated inside a sequence of nested lists with two elements (we use '(' and '[' to denote the introduction of `a` into a list). When we exit the function, `a` is extracted from a sequence of lists (we use ')' and ']' to denote the extraction of `a` from a list). In both cases, '(' and ')' represent the first position of the list (index 0) and '[' and ']' represent the second position of the list (index 1). Therefore, the structure in line 8: `a = [0, [[0, a], 0]]` encodes the string "[([". Similarly, in line 10: `a = a[1][1][0]` encodes the string "]])".

The interesting point is that question Q1 can only be answered *yes* when the sequence of operations applied to `a` ('(', '[', ')', ']') by a specific ordering and number of function calls belongs to $L(balanced)$. Additionally, the placement of the operations in functions and the call stack's behaviour guarantee that the order of operations is equivalent to a sequence $x_1 x_2 \dots x_n y_n \dots y_2 y_1$, where the subindex is each function's number.

Because the encodings produced by this program are exactly the same as in the P-PCP instance of Example 7.2 (their representation is exactly the same, see Fig. 7.2), and solving the RDA would be equivalent to finding a solution to the P-PCP instance, then answering correctly Q1 is undecidable for class C. Additionally, because *any instance* of P-PCP can be encoded (using the same scheme) in a program like this, which requires a context-sensitive field-sensitive analysis to correctly answer the question, Reps concluded that RDA is undecidable for programs in class C.

It is important to remark that the program in Figure 7.3 encodes a P-PCP instance for one specific analysis: F$^+$C$^+$RDA. Note that if we answer *no* to question Q1 (meaning that $(23, a)$ cannot be equal to 42), this does not imply that the value of a at line 23 does not depend on the value of a at line 1. In fact, there are many possible executions in which $(23, a)$ data depends on $(1, a)$: all those in which the value of $(23, a)$ contains the value 42. For instance, in the execution `f()` $\rightarrow$ `f1()` (with the trace: $\langle 1, 22, 17, 18, 2, 3, 4, 5, 23 \rangle$), the value of $(23, a)$ is $[0, [42, 0]]$. Therefore, $(23, a)$ data depends on $(1, a)$, but this execution is not a solution to the P-PCP instance. This clearly shows that Reps' results are bounded to a specific analysis: RDA. Specifically, Reps proved that F$^+$C$^+$RDA is undecidable for class C. But, this result cannot be applied to

```
1  a = 42  # A unique value that does not appear again
2  stack = []
3  first = True
4  while first or cond1:
5    first = False
6    if cond2:
7      a = [0, [[0, [a, 0]], 0]]  # x1: ([([
8      stack.append(1)
9    else if cond3:
10     a = [0, [[0, a], 0]]       # x2: [([
11     stack.append(2)
12   else:
13     a = [0, [0, [0, a]]]       # x3: [[[
14     stack.append(3)
15 while stack != []:
16   if   stack[-1] == 1:
17     a = a[1][0]                # y1: ])
18   elif stack[-1] == 2:
19     a = a[1][1][0]             # y2: ]])
20   else: # 3
21     a = a[1][0][1][1][1][1][0] # y3: ])]]]])
22   stack.pop()
23 # Q1: Could a be 42 here?
```

FIGURE 7.4: A Python program whose F^-C^+RDA is equivalent to solving the P-PCP instance from Example 7.2.

other classes of programs, or to other analyses (e.g., DFA and DDA).

7.3.2 RDA without functions and with composite data structures (F^-C^+RDA)

In this section we follow the same approach as in the previous section. Similar to Example 7.3, we identify a program for which solving a given RDA is equivalent to solving the P-PCP instance from Example 7.2. The main difference is that we use a program in class E, there are no function calls, and thus only intraprocedural analyses can be performed on it. Therefore, with this change, context-sensitive and context-insensitive analyses are equivalent. This shows that the same problem identified by Reps appears even when the analysis is context-insensitive (thus, both context-sensitive and context-insensitive analyses are undecidable for class E). As we still include composite data structures (the analyses can be field-sensitive) they are F^-C^+RDA.

Example 7.4 (Counterexample for F$^-$C$^+$RDA undecidability)**.**
Consider the program in Fig. 7.4, which contains two sequential `while` loops. In the first loop, `a` is placed at an arbitrary position of a data structure formed by nested lists of two elements. In every iteration, one of the branches is selected by the conditionals (lines 6, 9, and 12) and the index of the encoding used is pushed to a stack (1, 2, or 3 for x_1, x_2, and x_3; according to the terminology used in Example 7.2). In the second `while` loop, the stack is used to encode y_1, y_2, and y_3 in the reverse order (see the conditionals in lines 16, 18, and 20; and the `pop()` in line 22). $Q1$ can only be answered *yes* (`a = 42` in line 23) if every nesting operation and extraction match exactly, such that the solution $x_{i_1}x_{i_2}\dots x_{i_n}y_{i_n}\dots y_{i_2}y_{i_1}$ is guaranteed to belong to $L(balanced)$.

Note that the encoding of each x and y is the same as in Example 7.3 (e.g., line 19 in Fig. 7.4 is identical to line 10 in Fig. 7.3, both encode y_2). Additionally, the `stack` variable and both loops work together to simulate the call stack in that same example. Thus, this program is equivalent to the one in Fig. 7.3, with the caveat that no context-sensitive analysis can be performed on it because there is only one procedure (it is intraprocedural code with no calls).

Like its equivalent from the previous subsection, this program encodes a P-PCP instance. Question $Q1$ remains the same, and so does its answer (*yes*). However, this answer can only be obtained if an execution path is found that solves the P-PCP instance. In this program, `a` can be 42 at the end when the first loop is traversed four times, and the `stack`'s value after the first loop is `[1,2,3,1]` (a solution to the P-PCP instance). This corresponds with the following execution path:

$$\langle 1,2,3,4,5,6,\overset{x_1}{7},8,4,5,6,\overset{x_2}{9},10,11,4,5,6,9,\overset{x_3}{12},13,14,4,5,6,\overset{x_1}{7},8,15,16,\overset{y_1}{17},22,15...\rangle$$

The sequence of values of variable a, after applying each operation, is:

$$\begin{aligned}
a &: 42\\
x_1 &: [0,[[0,[42,0]],0]]\\
x_2 &: [0,[[0,[0,[[0,[42,0]],0]]],0]]\\
x_3 &: [0,[0,[0,[0,[[0,[0,[[0,[42,0]],0]]],0]]]]]\\
x_1 &: [0,[[0,[[0,[0,[0,[0,[[0,[0,[[0,[42,0]],0]]],0]]]]]],0]],0]]\\
y_1 &: [0,[[0,[0,[0,[0,[[0,[0,[[0,[42,0]],0]]],0]]]]]],0]]\\
y_3 &: [0,[0,[[0,[42,0]],0]]]\\
y_2 &: [0,[42,0]]\\
y_1 &: 42
\end{aligned}$$

An important consideration about our program transformation (replacing functions by `while` loops; i.e., replacing recursion by iteration) is the following: semantically, `while` loops can be defined as recursive calls with a single caller (their location in the program), and, thus, we could consider that Example 7.4 contains functions in the form of the loops. However, because context-sensitivity requires more than one call per function (as it differentiates between calls to a function), there is no difference between a context-sensitive analysis and a context-insensitive analysis in our transformed program. Therefore, our transformation introduces a fundamental semantic implication: context-sensitive and context-insensitive analyses become equivalent. This transformation allows us to prove that undecidability is reached with simpler programs (without functions).

7.3.3 RDA without composite data structures

As with previous sections, Reps' example can be further adapted to replace complex data structures with any infinite-domain data type, encoding a nested array with digits on an arbitrarily large integer or string, and a stack by "pushing" and "popping" values by multiplying and dividing by the number's base. However, unlike the transformation of functions into loops and a supporting stack, this transformation does not fully remove the need for a complex data structure.

Example 7.5 (Encoding complex data structures into strings).
Consider the sequence "[([", which in previous examples corresponded to x_2, and was represented as a nested array: the current value was placed in the second position, then the first and then the second position of an array, i.e., `a = [0, [[0, a], 0]]`. We can represent this array as a string instead, by encoding each nested array as a character. The initial value could be `"42"`, and as it is pushed further into a nested array, elements are added at the end, indicating its location. For example, if we were to use `"["` and `"("`, the string after applying x_2 would be `"42[(["`.

Most computer scientists will not consider the string a basic data type, as they are a sequence or array of characters; but we can also use integers to encode composite data structures, with a small caveat:

Example 7.6 (Encoding complex data structures into integers).
Instead of using strings, the same problem can be encoded into integers: placing a value a in the n-th index of an array would be equivalent to $10a + n$, so placing the value 42 in the second position of an array (index 1) would be $10 \times 42 + 1 = 421$. If we have the sequence "[([" (x_2), we can

encode it as $n_1 = 1$, $n_2 = 0$, $n_3 = 1$. Hence, if we apply $10a + n_i$ for all three positions, we have:

$$a_1 = 10a + n_1 = 10 \times 42 + 1 = 421$$
$$a_2 = 10a_1 + n_2 = 10 \times 421 + 0 = 4210$$
$$a_3 = 10a_2 + n_3 = 10 \times 4210 + 1 = 42101$$

This encoding cannot hold indefinitely, as integers are, in most programming languages, limited. There is a finite domain of integers, and reaching that limit causes an overflow error or a silent overthrow. In both cases, this means that arrays cannot be encoded into integers. However, there are some programming languages in which integers are unbounded, and thus this encoding is valid, such as *BigInteger* in Java, integers in Python or *Integer* in Haskell.

Through these two example encodings, we can see that complex data structures must be present in the program, either explicitly or implicitly, if we want it to encode a P-PCP instance. We have not been able to design a program for which RDA is undecidable without complex data structures, and that means that we cannot say for certain whether classes D and F are decidable or not.

7.3.4 Undecidability of data-dependence analysis

In the preceding sections we have showcased counterexamples to demonstrate the undecidability of Reps' data analysis (RDA) in programs with complex data structures. In this and the following section, we show that the same examples can be tweaked so that the question asked represents a DDA or even a DFA.

We prove the undecidability of DDA showing that all instances of P-PCP can be encoded in programs for which finding the solution to the DDA implies finding a solution to an instance of P-PCP. We can convert any of the RDA examples presented in the previous sections to an equivalent example for DDA. For instance, in Fig. 7.5 we can see a program based on the one presented in Example 7.3. Question Q1 can only be solved by solving an instance of P-PCP, and this happens for all DDAs.

To explain why answering Q1 is undecidable we can consider the truth table shown in Table 7.1. It has one row for each possible state that a can have at the end of the program:

1. The end of the program is not reached, due to an exception being raised by an invalid unpacking (a is `["42", 0]`, and an operation like `a[1][0]`

```
1  a = "42" # A value, unique in the program
2  def f1():
3    a = [0, [[0, [a, 0]], 0]]  # x1: ([([
4    if cond1: f()
5    a = a[1][0]                # y1: ])
6
7  def f2():
8    a = [0, [[0, a], 0]]       # x2: [([
9    if cond2: f()
10   a = a[1][1][0]             # y2: ]])
11
12 def f3():
13   a = [0, [0, [0, a]]]       # x3: [[[
14   if cond3: f()
15   a = a[1][0][1][1][1][1][0] # y3: ])]]]])
16
17 def f():
18   if   cond4: f1()
19   elif cond5: f2()
20   else:     f3()
21
22 f()
23 if type(a) is not str: a = 41
24 # Q1: Does (24,a) depend on (1,a)?
```

FIGURE 7.5: An undecidable F^+C^+DDA

is applied). In P-PCP terms, there is an unmatched closing symbol in the sequence, such as '(]', which means that the sequence is not a valid solution. The P-PCP instance is not solved and the DDA would answer *no* to Q1 (because Q1 is not reachable at all).

2. a is "42", as the sequence of packing and unpacking operations match exactly. In P-PCP terms, the sequence would be a word in $L(balanced)$, and thus a valid solution. The P-PCP instance is solved and the DDA would answer *yes* to Q1 because `type(a)` is a string (a = "42").
3. At line 23, `type(a) is not str` holds, and consequently, `a=41`. This can only happen because a is a list (e.g., `[0,a]`) or it is an integer (`0`). In the P-PCP, the word is not in $L(balanced)$ due to missing closing symbols, e.g. '([]'. The P-PCP instance is not solved and the DDA would answer *no* to Q1 (because (24,a) depends on (23,a) but not on (1,a)).

Therefore, answering *yes* to Q1 is equivalent to solving the P-PCP instance. Because any instance of P-PCP can be coded in the same way, and because P-PCP is undecidable, so is the DDA analysis to answer Q1 for class C. The same

TABLE 7.1: A truth table showing the equivalence of the DDA in Figure 7.5 to the P-PCP instance from Example 7.2.

Is Q1 reached?	DDA(Q1)	P-PCP
(1) No, extraction error	No	Unsolved
(2) Yes, a is "42"	Yes	Solved
(3) Yes, a is 41	No	Unsolved

conditional statement in line 23 can be added to the program in Example 7.4 to prove that DDA is also undecidable for class E.

7.3.5 Undecidability of data-flow analysis

The undecidability of DFA can be proved with the same strategy used for DDA. We need to implement a program for which the DFA analysis answers *yes* to a question if and only if the P-PCP instance has a solution. This program is shown in Figure 7.6.

Note that this program uses an additional variable `b`, and the DFA analysis' question is whether the definition of `b` at line 1 can reach the end of the program. The truth table generated by this program for the analysis DFA is analogous to Table 7.1. The three possible cases when reaching question Q1 are:

1. An exception is raised due to an invalid unpacking (in P-PCP terms, there is an unmatched closing symbol in the sequence, such as '(]'). In this case, the P-PCP is not solved and the DFA should answer *no* because Q1 is not reached due to the exception.
2. `a` is "42", as the sequence of packing and unpacking operations match exactly (in P-PCP terms, the sequence would be a word in $L(balanced)$). In this case, a solution to the P-PCP has been found and, thus, (1,`b`)'s definition reaches the end (because `b` $= 6$ is not executed), so a DFA would answer *yes* (`b` $= 7$).
3. `a` is a not a string in line 24 (it is either a list or `0`). This can only happen if `a` has not been completely unpacked (in P-PCP terms, the word is not in $L(balanced)$ due to missing closing symbols, e.g. '([]'). In this case, the condition in line 24 holds and `b` is overwritten (`b` $= 6$). The P-PCP instance is not solved and the DFA should answer *no* because the initial value of `b` has been killed in line 24 (`b` $= 6$).

Here again, answering *yes* to Q1 is equivalent to solving the P-PCP instance. Since all P-PCP instances can be codified in the same way, and because P-PCP

```
1  b = 7
2  a = "42" # A unique value not appearing again
3  def f1():
4    a = [0, [[0, [a, 0]], 0]]  # x1: ([([
5    if cond1: f()
6    a = a[1][0]                # y1: ])
7
8  def f2():
9    a = [0, [[0, a], 0]]       # x2: [([
10   if cond2: f()
11   a = a[1][1][0]             # y2: ]])
12
13 def f3():
14   a = [0, [0, [0, a]]]       # x3: [[[
15   if cond3: f()
16   a = a[1][0][1][1][1][1][0] # y3: ])]]]])
17
18 def f():
19   if   cond4: f1()
20   elif cond5: f2()
21   else:     f3()
22
23 f()
24 if type(a) is not str: b=6
25 # Q1: has (1,b) reached this line?
```

FIGURE 7.6: An undecidable F^+C^+DFA

is undecidable, so is any F^+C^+DFA used to answer Q1. The same lines 1 and 24 can be added to the program in Example 7.4 to also prove that DFA is undecidable for class E.

These program transformations show that the original examples used to prove that Reps' data analyses are undecidable can be used (after the transformation) to prove the undecidability of data-dependence and data-flow analyses. We can conclude that all three kinds of data analyses (RDA, DDA, and DFA) are undecidable for classes C and E.

Chapter 8

Context- & Field-sensitive Program Slicing

> A person is an individual, [...] but a person can only evolve in the context of a species capable of forming societies.
>
> *Lindsay Ellis, Truth of the Divine*

As described in Section 2.3, program slicing can be context- and/or field-sensitive, gaining precision with each sensitivity acquired. Context-sensitivity is a long-solved problem for dependence graphs, as it is the main feature of the SDG (introduced more than 35 years ago [52, 53], solving the problem with a linear-cost traversal). Field-sensitivity presents a greater challenge, as most techniques increase the complexity of slicing to polynomial or sometimes exponential; and they are difficult to combine with other context-sensitive techniques, such as the SDG.

In this chapter, we explore the constrained-edges PDG, a field-sensitive graph that uses labels and a stack (Section 8.1); and tabular slicing, a interprocedural dataflow analysis technique that can be adapted to produce context-sensitive slices (Section 8.2). Then, we adapt both such that they can be combined into a single technique, that is both context- and field-sensitive (Section 8.3). Finally, we perform an empirical evaluation on this new technique, with an implementation for Erlang programs (Section 8.4) and compare against other approaches and techniques (Section 8.5).

8.1 The constrained-edges PDG

The constrained-edges PDG (CE-PDG) [37] is an extension to the PDG that decomposes data structures into separate nodes and adds labels to each arc in the graph. It treats these labels as constraints during slicing, such that only

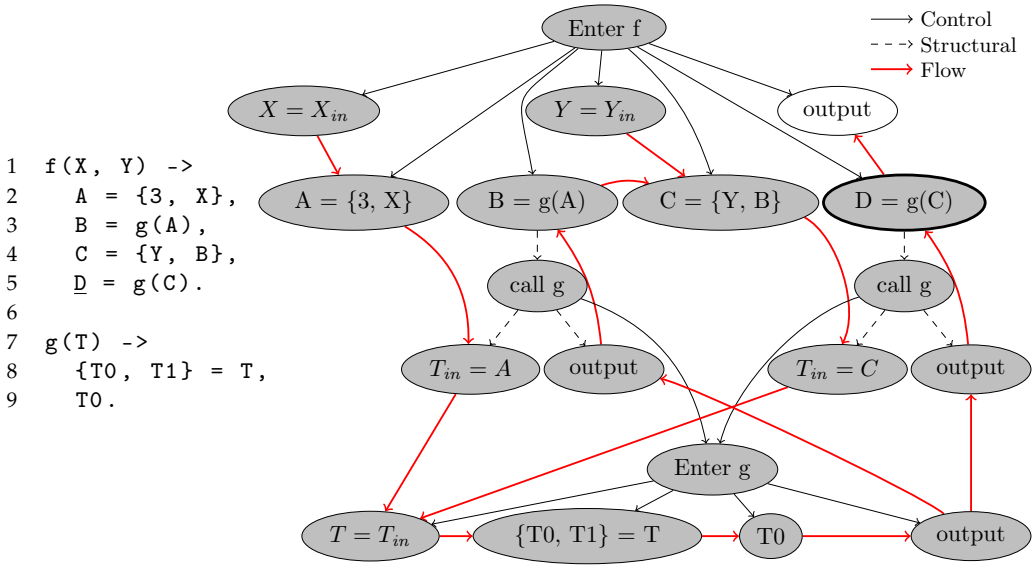


FIGURE 8.1: An Erlang program that requires a context- and field-sensitive analysis to obtain the minimal slice w.r.t. $\langle 5, \texttt{D} \rangle$, and its PDG, with the criterion in bold and a (non-field-sensitive) slice in grey.

the specific elements of a data structure that are necessary will be included in any given slice. This section gives a brief introduction to its features and the algorithms used to slice it, so readers interested in all the details should consult [37].

Example 8.1 (The need for context- and field-sensitive slicers)**.** Figure 8.1 shows a simple Erlang program, which packs and unpacks data from tuples. Some of these actions are performed at the same level, while others are "hidden away" in function g. Its PDG is shown in the same figure, and the (non-field-sensitive) slice produced w.r.t. $\langle 5, \texttt{D} \rangle$ is the whole program. However, the reader can realise that g only needs the first element of the tuple, and thus the assignments of `A` and `B` are unnecessary, because `C`'s assignment uses `B` in its second element. We use this example throughout the chapter to illustrate the improvements made by each slicing algorithm shown.

8.1.1 Building the CE-PDG

Any PDG can be transformed into a CE-PDG in three steps: (i) split nodes that contain data structures, (ii) add flow dependences between the new nodes, and

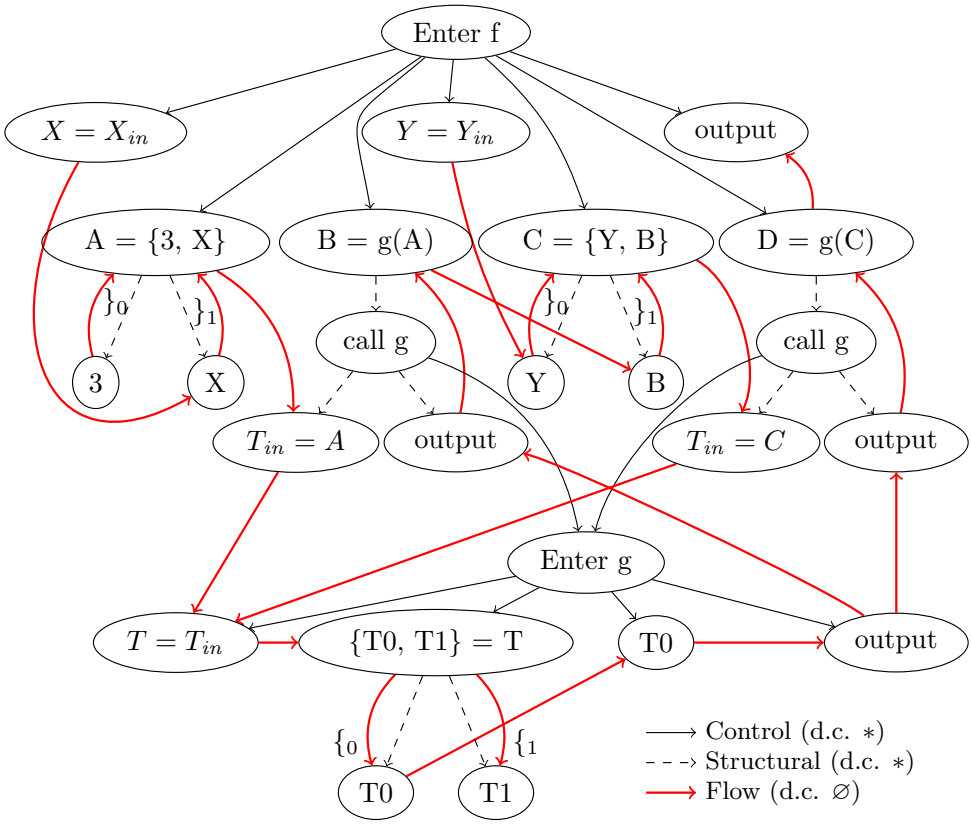


FIGURE 8.2: The CE-PDG for the program in Fig. 8.1.

(iii) label all arcs with constraints. We illustrate these changes by comparing Fig. 8.1 with Fig. 8.2 in each step.

Split nodes. Each data structure (e.g., list, array, tuple, record, object) that appears in the source code is decomposed into an AST-like structure, using the so-called structural dependences to connect each node to its child elements. Each data structure's root is the statement from which it was decomposed. Structures are decomposed according to the source code's AST, producing a finite number of nodes. In our case, the new nodes appear below `A` and `C`'s assignments, and at the bottom, at the definition of `T0` and `T1`.

Add flow dependences. Once a data structure has been split into nodes, flow dependences whose source or target has been affected by the previous step must be moved accordingly, and new flow dependences must be created to link the data structure's elements. Definitions of data structures will have

flow dependences in parallel to structural arcs, and usages will produce flow dependences in the opposite direction. In Fig. 8.2, the flow dependence from `B = g(A)` to `C = {Y, B}` has moved to `B = g(A)` $\rightarrow$ `B`, and the one from `{T0, T1} = T` to `T0` is now `T0` $\rightarrow$ `T0`. Among the new flow dependences we can find `X` $\rightarrow$ `A = {3, X}` and `{T0, T1} = T` $\rightarrow$ `T1`.

Label arcs. There are three types of labels or constraints: empty ($\varnothing$), asterisk ($*$) and access constraint (op_{pos}). For the latter, *op* indicates the type of data structure being accessed and whether it constitutes a definition or usage. We use '{', '[', '}', and ']' to denote a tuple's definition, a list's definition, a tuple's usage, and a list's usage, respectively. The *pos* indicates the position of the element being accessed in the tuple or list, and can be either an integer index, *H* (head of a functional list) or *T* (tail).

Let's explore the behaviour of each constraint by seeing which arcs are labelled[1] with them:

Flow dependences can be labelled with empty ($\varnothing$) or access ($\{_a, \}_b, [_c,]_d$) constraints. Flow dependences that connect a definition to a usage of the same variable are labelled with empty constraints. These do not have any effect during traversal, because no complex data structure is being manipulated, a value has been defined and used as-is. For example, the arc `X = X`$_{in}$ $\rightarrow$ `X` shows no label (the default for flow arcs is the empty constraint).

However, flow dependences introduced in the previous step, between elements of a data structure, are labelled with access constraints. As we have mentioned, definitions use open symbol labels ('{', '['), and usages, the closing variants ('}', ']'). Access constraints will be used to match the position of definitions on usages, pairing matching opposing constraints. In our graph, `3` and `X` are connected to their parent (`A = {3, X}`), and they represent the usage (closing) of the first (0) and second (1) elements of a tuple (curly bracket). Thus, their flow arcs are labelled with $\}_0$ and $\}_1$.

Finally, any other non-flow dependences, such as control and structural dependences, are labelled with asterisk ($*$) constraints. This constraint acts as a *clear stack* operation, which forgets all the context that has been stored, because once one of these arcs is traversed, the sequence of flow arcs has been broken. That is to say, when we traverse a control dependence, the source of that arc is needed, and so are all the data structures that are attached to it, regardless of previous access constraints.

$$
\begin{aligned}
S &::= C\,O \\
C &::= \}_i\,C \mid]_p\,C \mid R\,C \mid \varnothing\,C \mid *\,S \mid \epsilon \\
R &::= \{_i\,R\,\}_i \mid [_p\,R\,]_p \mid \varnothing\,R \mid \epsilon \\
O &::= \{_i\,O \mid [_p\,O \mid R\,O \mid \varnothing\,O \mid *\,S \mid \epsilon
\end{aligned}
$$

FIGURE 8.3: Valid constraint sequences grammar.

8.1.2 Labels as constraints

We have already given a short description on how arcs' labels act as constraints during slicing. When traversing labelled arcs, the sequence formed by the labels must be in the language $L(S)$, defined by the grammar in Fig. 8.3. Otherwise, the sequence of arcs is not traversable. Given a valid sequence, any prefix, suffix, or infix of that sequence is also valid. Asterisk and empty constraints can be easily understood from their role in the grammar: asterisk constraints only appear before the grammar restarts anew ($*\,S$), and empty constraints can be placed in any location ($\varnothing\,C$, $\varnothing\,R$, $\varnothing\,O$). However, access constraints are more limited. A valid sequence consists of a series of closing constraints, followed by opening constraints; any of the two can be empty (ϵ), and they can contain any number of interleaved *resolved* constraints. These are any number of opening and closing constraints, as long as they match each other, i.e., "resolve" each other.

Example 8.2 (Valid and invalid constraint sequences)**.**
In the running example's CE-PDG (Fig. 8.2), consider the sequence of arcs from `T0` to `3`:

$$\texttt{T0} \xrightarrow{\{_0} \texttt{\{T0, T1\} = T} \xrightarrow{\varnothing} \texttt{T = } \mathtt{T}_{in} \xrightarrow{\varnothing} \mathtt{T}_{in} \texttt{ = A} \xrightarrow{\varnothing} \texttt{A = \{3, X\}} \xrightarrow{\}_0} \texttt{3}$$

It forms the sequence of constraints $\{_0\,\varnothing\,\varnothing\,\varnothing\,\}_0$, which does belong to $L(S)$: $S \to C\,O \to O \to R\,O \to R \to \{_0\,R\,\}_0 \to \{_0\,\varnothing\,R\,\}_0 \to \{_0\,\varnothing\,\varnothing\,R\,\}_0 \to \{_0\,\varnothing\,\varnothing\,\varnothing\,R\,\}_0 \to \{_0\,\varnothing\,\varnothing\,\varnothing\,\}_0$.

However, if we change the last arc (labelled $\}_0$) to `A = {3, X}` $\xrightarrow{\}_1}$ `X`, the new sequence ($\{_0\,\varnothing\,\varnothing\,\}_1$) is no longer valid: $S \to C\,O \to O \to \{_0\,O \to \{_0\,\varnothing\,O \to \{_0\,\varnothing\,O \to \{_0\,\varnothing\,\varnothing\,O \to \{_0\,\varnothing\,\varnothing\,\varnothing\,O$ (it is not possible to make O produce $\}_1$).

[1] In Fig. 8.2, to simplify the graph, only access constraint labels are shown. All other arcs' labels are shown in the graph's key (d.c. stands for default constraint).

TABLE 8.1: Rules to process a constraint with a given stack.

Case	Input	Constr.	Output
1.	S	$\varnothing$	S
2a.	S	$\{_x$	$S\{_x$
2b.	S	$[_x$	$S[_x$
3.	$\bot$	$\}_x$ or $]_x$	$\bot$
4a.	$S\{_x$	$\}_x$	S
4b.	$S[_x$	$]_x$	S
5.	S	$*$	$\bot$

Nevertheless, the grammar is not suitable during slicing, as with every arc traversed, we need to check whether the sequence of constraints belongs to $L(S)$. Instead, during slicing we accumulate open access constraints ('$\{_a$', '$[_b$') on a stack, and stop the traversal whenever an invalid closing access constraint is found, stopping the traversal at that arc. Table 8.1 shows a summary of the resulting stacks (output) when a constraint is applied to a stack (input). In the table, S stands for any stack, $\bot$ is the empty stack, and x is any position indicator (an integer, H or T). In rows with multiple appearances of x, it represents the same value throughout the row. Any combination that is not present on the table is invalid, and the arc should not be traversed. All invalid combinations appear with a closing access constraint that does not match an open constraint at the top of a non-empty stack.

Example 8.3 (Verifying sequences with Table 8.1)**.**
Consider the sequences from the previous example, $\{_0 \varnothing \varnothing \varnothing \}_0$ and $\{_0 \varnothing \varnothing \varnothing \}_1$, and a variation, $\{_0 \varnothing * \varnothing \}_1$. Let us analyse them with a stack:

$$\bot \xleftarrow{\{_0} \{_0 \xleftarrow{\varnothing} \{_0 \xleftarrow{\varnothing} \{_0 \xleftarrow{\varnothing} \{_0 \xleftarrow{\}_0} \bot$$

$$\bot \xleftarrow{\{_0} \{_0 \xleftarrow{\varnothing} \{_0 \xleftarrow{\varnothing} \{_0 \xleftarrow{\varnothing} \{_0 \xleftarrow{\}_1} \textit{error}$$

$$\bot \xleftarrow{\{_0} \{_0 \xleftarrow{\varnothing} \{_0 \xleftarrow{*} \bot \xleftarrow{\varnothing} \bot \xleftarrow{\}_1} \bot$$

The first succeeds, applying rules 2a, 1 and 4a; the second stops before the last constraint, as $\}_1$ is not the complementary access constraint to $\{_0$; and the third succeeds thanks to the asterisk constraint (applying, in order, rules 2a, 1, 5, 1, 3).

Algorithm 7 Constrained slicing algorithm for the CE-PDG (simplified) [37].

```
1:  function SLICE(G = (N, A), sc)
2:      processed ← ∅
3:      workList ← {⟨sc, ⊥, _⟩}        ▷ Triplets with a node, stack and arc type.
4:      while workList ≠ ∅ do
5:          ⟨n, s, t⟩ ← POP(workList)
6:          for all (n′, n) ∈ A do
7:              t′ ← ARCTYPE((n′, n))
8:              if ¬(t = structural ∧ t′ = flow) then
9:                  s′ ← PROCESSCONSTRAINT(s, (n′, n))
10:                 if s′ ≠ error ∧ ⟨n′, s′, t′⟩ ∉ processed then
11:                     workList ← workList ∪ {⟨n′, s′, t′⟩}
12:         processed ← processed ∪ {⟨n, s, t⟩}
13:     return {n | ⟨n, s, t⟩ ∈ processed}
14: function PROCESSCONSTRAINT(s, e)
15:     c ← ARCCONSTRAINT(e)
16:     ⟨op, pos⟩ ← c
17:     if c = AsteriskConstraint then return ⊥                    ▷ Case 5
18:     else if c = EmptyConstraint then return s                  ▷ Case 1
19:     else if op = { ∨ op = [ then PUSH(c, s)                    ▷ Case 2
20:     else if s = ⊥ then return ⊥                                ▷ Case 3
21:     else if op = } ∧ TOP(s) = ⟨{, pos⟩ then POP(s)             ▷ Case 4a
22:     else if op = ] ∧ TOP(s) = ⟨[, pos⟩ then POP(s)             ▷ Case 4b
23:     else return error                  ▷ Error, no case has been matched.
24:     return s
```

8.1.3 Slicing the CE-PDG

The CE-PDG slicing algorithm is a modification of the basic slicing algorithm, including a stack alongside every node visited, and a method to check whether a constraint can be applied on top of a given stack (and what resulting stack would be). The full constrained slicing algorithm also includes efficiency considerations by introducing a partial ordering between stacks, and specific handling for flow dependence cycles, which may cause an uncontrolled growth of the stack. However, since both problems disappear when the algorithm is combined with tabular slicing, we have excluded them from the algorithm in this chapter.

A simplified version appears on Algorithm 7. It uses a *workList* to store triplets, each containing the node and stack reached, along with the last arc visited. The latter is relevant because structural dependence introduces a

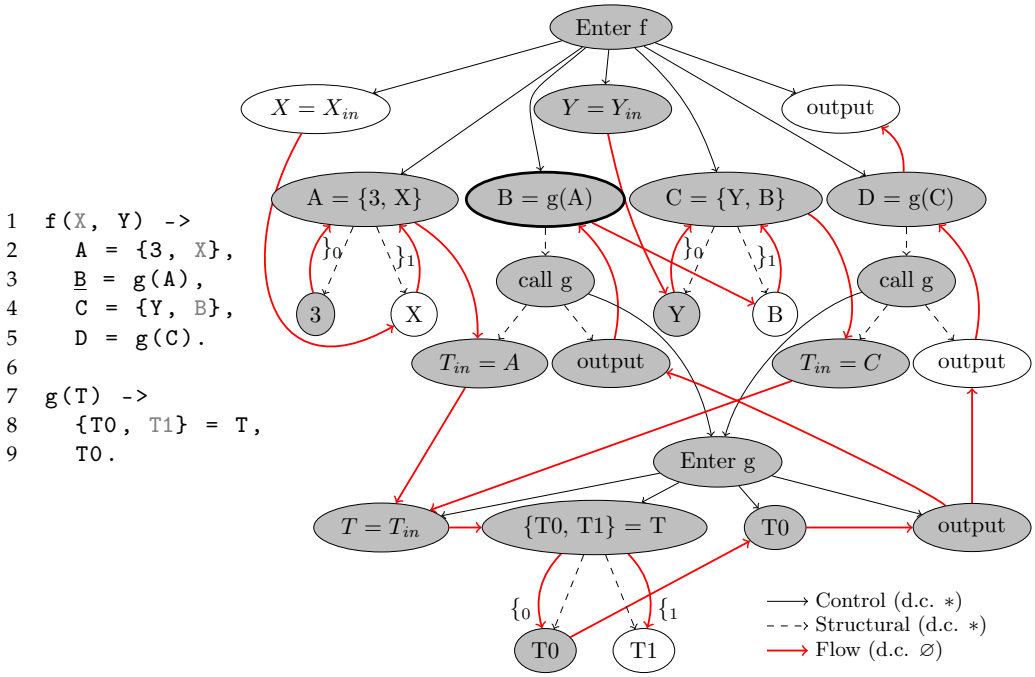


FIGURE 8.4: Slice of Fig. 8.2 w.r.t. $\langle 3, \texttt{B} \rangle$ (Algorithm 7).

traversal limitation: flow dependences must not be traversed immediately after structural arcs[2]. Then, it traverses arcs backwards, applying the restrictions on the stack with PROCESSCONSTRAINT, which applies Table 8.1's logic. At the end of the algorithm, the set of nodes present in *processed* is the slice.

The maximum number of elements in *processed* $\cup$ *workList* at any moment is $2|N|(s^d + 1)$, where there are two arc types, N is the set of nodes in the graph, s is the number of different open access constraints that appear in the graph, and d is the maximum depth reached by the stack. Thus, the temporal asymptotic cost of this algorithm is $O(s^d n)$.

Example 8.4 (Slicing Fig. 8.2 with Algorithm 7)**.**
Figure 8.4 shows the field-sensitive slice of our running example. Compared to the PDG, five elements have been removed: one is unreachable (`T1`); two have been excluded because no valid stack could reach them, `X` and `B` (the stack was $\{_0$ on `A = {3, X}` and `C = {Y, B}`, respectively); one was excluded transitively (`X` = $\texttt{X}_{in}$ is excluded because `X` was excluded); and

[2]Although it is not specified in the algorithm, all other arc types can be merged into a single type for efficiency, such that the possible values for the triplet's last element are *structural* and *other*. Right before line 11, if $t' \neq$ *structural*, then $t' \leftarrow$ *other*.

one was left out thanks to the structural, then flow traversal restriction (`B = g(A)` was reached via structural dependence, the flow arc `output` → `B = g(A)` is not traversed).

This example is both good and bad news. Some elements have been excluded, and even a function parameter has been removed. However, without context-sensitivity, the dependence on the first call (`B = g(A)`) meant that the other call to `g` (`D = g(C)`) was included unnecessarily.

8.2 Tabular slicing

Summary arcs in the SDG must be precomputed, so that the graph can achieve a linear time complexity when extracting slices. However, in the CE-PDG, we cannot simply decide whether there is a summary arc between an actual-in and an actual-out. Instead, we must consider every possible stack in both, and then consider whether there is a summary arc between each pair actual-in × stack, actual-out × stack. With an unbounded stack, the possibilities are infinite. Even with a maximum capacity, the growth in the number of summary arcs to be computed is exponentially. Thus, the SDG and its system for precomputing summaries is not compatible with the CE-PDG.

Lucky for us, there are techniques that can produce context-sensitive slices without precomputing summaries, such as tabular slicing, a context-sensitive slicing technique that was originally defined as a strategy to make dataflow analyses interprocedural [94]. When applied to slicing, it behaves similarly to the SDG, but computes summary arcs on-the-fly instead of incorporating them into the graph. Even though its temporal efficiency (quadratic) is worse than the classic slicing algorithm (which is linear), it speeds up the construction of the graph[3]. In some cases, such as large graphs from which small slices are extracted, it can be overall faster (taking into account construction and traversal) than the classic algorithm.

Algorithm 8 contains the tabular slicing algorithm, adapted for backwards slicing. It uses three data structures to store nodes during slicing.

pathEdge[4] The titular table, a set of node pairs. The first node in the pair is the *current* node that has just been reached, and the second is its *context*, a formal-out node through which the current function was entered. The context may not exist (often represented as *null*, noted here as ∘), which means that no caller ← callee dependence has been traversed (although caller → callee might have been traversed). When comparing to the

[3]The asymptotic complexity of building a SDG remains quadratic.

Algorithm 8 Tabular slicing algorithm (adapted from [94, Fig. 3])

Input: $G = (N, A)$ is a PDG, A_{out} $N_{in}, N_{out} \subset N$ are actual-in/out nodes.

```
 1: function SLICE(G = (N, A), sc)
 2:     pathEdge ← ∅                          ▷ Contains pairs ∈ N × (N ∪ {∘}).
 3:     workList ← ∅                          ▷ Contains pairs ∈ N × (N ∪ {∘}).
 4:     summaries ← ∅                         ▷ Contains summary arcs ∈ N_in × N_out.
 5:     PROPAGATE(⟨sc, ∘⟩)
 6:     while workList ≠ ∅ do
 7:         ⟨n, n'⟩ ← POP(workList)
 8:         if ISENTER(n) then
 9:             TRAVERSEARCS(⟨n, n'⟩, n' ≠ ∘)
10:         else if ISACTUALOUT(n) then
11:             TRAVERSEARCS(⟨n, n'⟩, true)
12:             for all (n_in, n) ∈ summaries do          ▷ Follow summaries.
13:                 PROPAGATE(⟨n_in, n'⟩)
14:             for all n_fo | (n_fo, n) ∈ A_out do       ▷ n_fo: formal-out node.
15:                 PROPAGATE(⟨n_fo, n_fo⟩)
16:         else if ISFORMALIN(n) then
17:             TRAVERSEARCS(⟨n, n'⟩, n' ≠ ∘)
18:             if n' ≠ ∘ then                            ▷ New summary found.
19:                 for all ⟨n_in, n_out⟩ ∈ ACTUALNODES(n, n') do
20:                     if (n_in, n_out) ∉ summaries then
21:                         for all (n_out, n'') ∈ pathEdge do
22:                             PROPAGATE(⟨n_in, n''⟩)    ▷ Apply retroactively.
23:                         summaries ← summaries ∪ {(n_in, n_out)}   ▷ Store arc.
24:         else TRAVERSEARCS(w, true)
25:     return {n | ⟨n, n'⟩ ∈ pathEdge}
26: function PROPAGATE(e = ⟨n, n'⟩)
27:     if e ∉ pathEdge then
28:         pathEdge ← pathEdge ∪ {e}
29:         PUSH(e, workList)
30: function TRAVERSEARCS(⟨n, n'⟩, intraOnly)
31:     for all (n'', n) ∈ A do
32:         if ¬ (intraOnly ∧ ISINTERPROCEDURAL((n'', n))) then
33:             PROPAGATE(⟨n'', n'⟩)
```

SDG's two pass algorithm, a non-existent context is equivalent to Phase 1, and all others, to Phase 2.

workList A list of pending items, each a node pair. This list is initialised with the pair $\langle sc, \circ \rangle$, the slicing criterion with no context. New elements are added when they are found, using *pathEdge* to avoid visiting the same node pair twice (with the PROPAGATE function).

summaries A set of summary arcs that have been found, represented as pairs of actual-in/out nodes.

During slicing, the algorithm picks elements from *workList* and traverses its incoming intraprocedural arcs backwards (line 24). In each traversal, newly found nodes are added to *pathEdge* and *workList*, and their context is copied from the previous node (as can be seen in every call to PROPAGATE, except for line 15). The three kinds of nodes that can have incoming interprocedural arcs (i.e., can cross function bounds) require specialised logic:

Enter Similar to the SDG, when the context is empty (Phase 1), interprocedural input arcs are traversed, and, when it is not (Phase 2), they are ignored (lines 8-9).

actual-out When an actual-out node is reached, its incoming arcs are traversed (line 11), including arcs in *summaries* (line 12), and the context is switched (line 14). The context switch is the specialised traversal of its incoming output arcs. These arcs have a formal-out as a source, which becomes both the current and context node (when the pair (n_a, n_{ctxt}) is visited, the output arc $n_f \to n_a$ generates the pair (n_f, n_f)). This guarantees that the context is always set to the last formal-out found in the traversal (if any).

formal-in If a formal-in is reached without context, its arcs are traversed, as usual (line 17). However, if it is reached with context, this signals that a summary arc has been found (from the current node, a formal-in, to the context node, a formal-out). Thus, summary arcs for each matching actual-in/out pairs must be added to *summaries* (lines 18-23). If any affected actual-out has already been visited, the summary arc is traversed immediately (each pair $(n_{out}, n_{ctxt}) \in$ *pathEdge* generates a new pair (n_{in}, n_{ctxt})). As with the *Enter* node, its incoming input arcs are only traversed when the context is empty.

The "table" of *pathEdge* is built up throughout the slicing process, and at the end it contains at most $|N| \times |N_{fout}|$ elements, where N_{fout} is the set of formal-out nodes in the graph, which is roughly logarithmically smaller than N. Thus, assuming that the cost of auxiliary operations (like arc lookup) is cheap enough, the complexity of this process is $O(n \log n)$. However, if the

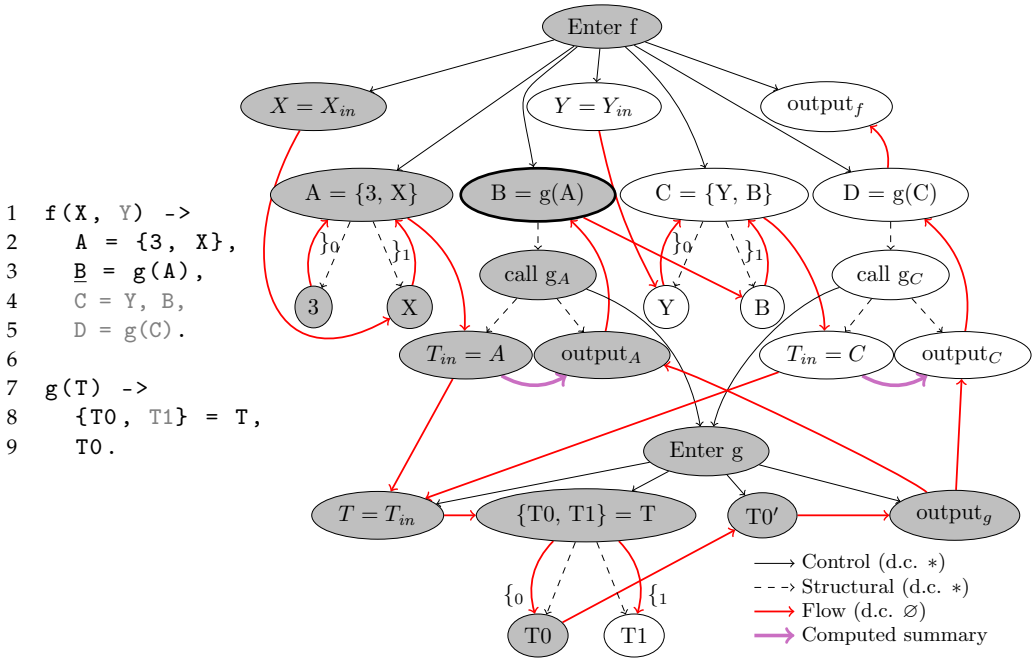


FIGURE 8.5: Slice of Fig. 8.2 w.r.t. $\langle 3, \mathtt{B} \rangle$ (Algorithm 8).

number of formal-out nodes in the graph rises, the cost comes closer to $O(n^2)$. At the end of the algorithm (line 25), the slice can be computed as the set of current nodes in the table (the first element in each tuple).

Example 8.5 (Slicing Fig. 8.2 with Algorithm 8)**.**
The completed tabular slice of our running example is shown on Fig. 8.5. Before analysing the results, let us step through the algorithm, one iteration at a time. To facilitate this process and avoid confusion, some nodes in the graph have been given a sub-index or a $'$. Table 8.2 details this process, showing a numbered row per iteration. In each iteration, one state is popped from the *workList*, obtaining its current and contextual nodes (*From* columns). Then, depending on the type of node, arcs are traversed (*Arc* column), and new states are reached (*Reached* column, one row per state reached). States that are reached but skipped are also shown, underlined in red. Each row is also annotated with the *Reason* why the node was reached and propagated (or skipped).

Every state that appears in *Reached* is added to *pathEdge* and *workList* (if it was not already present in *pathEdge*), and every state that appears in *From*

[5]Abbreviations for reasons. TE: TRAVERSEARCS call. CS: context switch (line 14). NS: new summary. EC: *Enter* node reached with a context node. NI: no incoming arcs.

TABLE 8.2: States explored and found in each iteration of the tabular slicing algorithm, to compute the slice at Fig. 8.5.

	From (POP)		Arc	Reached (PROPAGATE)		Reason[5]
	Current	Context		Current	Context	
0				B = g(A)	○	
1	B = g(A)	○	flow	out_A	○	TE
	B = g(A)	○	control	Enter f	○	TE
2	out_A	○	str.	call g_A	○	TE
	out_A	○	flow	out_g	out_g	CS
3	call g_A	○	str.	B = g(A)	○	TE
4	out_g	out_g	flow	T0′	out_g	TE
	out_g	out_g	str.	Enter g	out_g	TE
5	T0′	out_g	flow	T0	out_g	TE
	T0′	out_g	str.	Enter g	out_g	TE
6	T0	out_g	flow	{T0, T1} = T	out_g	TE
	T0	out_g	str.	{T0, T1} = T	out_g	TE
7	{T0, T1} = T	out_g	flow	T = T_{in}	out_g	TE
	{T0, T1} = T	out_g	str.	Enter g	out_g	TE
8	T = T_{in}	out_g	flow	T_{in} = A	○	NS
	T = T_{in}	out_g	flow	T_{in} = C	○	NS
	T = T_{in}	out_g	str.	Enter g	out_g	TE
9	Enter g	out_g	call	call g_A	○	EC
	Enter g	out_g	call	call g_C	○	EC
10	T_{in} = A	○	str.	call g_A	○	TE
	T_{in} = A	○	flow	A = {3, X}	○	TE
11	A = {3, X}	○	flow	3	○	TE
	A = {3, X}	○	flow	X	○	TE
	A = {3, X}	○	str.	Enter f	○	TE
12	3	○	str.	A = {3, X}	○	TE
13	X	○	str.	A = {3, X}	○	TE
	X	○	flow	X = X_{in}	○	TE
14	X = X_{in}	○	str.	Enter f	○	TE
15	Enter f	○	none			NI

is removed from *workList*. The first state reached is the slicing criterion, on row 0 (R0). Then:

R1. The first iteration follows arcs normally, reaching both "Enter f" and "output$_A$".
R2. Node "output$_A$" is an actual-out, so we perform a context switch to "output$_g$". We also reach "call g$_A$" via a structural arc.
R3. From "call g$_A$", we reach our first repeated node (so PROPAGATE does not alter any set).
R4. Starting on this row, we work with a context node ("output$_g$"), inside the function g. From its output node we reach "T0′" and "Enter g".
R5. "T0′" leads us to "Enter g" (another repeat) and to "T0".
R6. From "T0" we reach "{T0, T1} = T" twice (once via flow and another time via structural arc).
R7. "{T0, T1} = T" connects us to "Enter g" (reached for the third time) and to "T = T$_{in}$".
R8. We are at a formal-in ("T = T$_{in}$") *and* we have a context (the formal-out "output$_g$"), and thus we produce two summary arcs (one at each call, shown in Fig. 8.5). We immediately apply them retroactively: "T$_{in}$ = A" is reached via the summary at "call g$_A$"; but the other summary is not applied, as "output$_C$" has not been reached (and never will be). Additionally, we reach "Enter g" for the fourth time.
R9. Another special case: an *Enter* node ("Enter g") with context. In these cases, no interprocedural arcs must be traversed, so the two incoming arcs of this node are skipped.
R10. We are now processing states without a context. The actual-in node "T$_{in}$ = A" leads us to the definition of A ("A = {3, X}") and to a repeated node ("call g$_A$").
R11. From "A = {3, X}" we can reach two new nodes ("3" and "X") and a repeat ("Enter f").
R12. "3" is a dead-end: we only reach a "A = {3, X}", the node we came from.
R13. "X" helps us reach one of the formal-in nodes in this function: "X = X$_{in}$".
R14. From "X = X$_{in}$" we can only reach "Enter f" for the third time.
R15. Finally, "Enter f" has no incoming arcs, so no new state is obtained.

After iteration/row 15, the *workList* is empty, so the slicing process ends. Thanks to the restrictions in iterations 8 and 9, no element from the second call ("D = g(C)"), and thus half of the code in function `f` (and parameter `Y`) is not included in the slice.

This result can still be improved by field-sensitivity. With the combination of the tabular and constrained algorithms, we can produce a context- and field-sensitive slice: the minimal slice (in this case).

8.3 The context-sensitive CE-PDG

In this section, we combine the CE-PDG with tabular slicing to produce context- and field-sensitive slices. This is significant, because the SDG extension to the CE-PDG described in [88, Chapter 4.4], which precomputes summaries as grammars to be applied when slicing, cannot account for recursive calls. Due to the presence of the stack during the slice, precomputed summary arcs must take into account any possible input stack, which might be impossible if the stack size is not bound by a user-set limit.

The graph itself remains unchanged: the CE-PDG is built according to the description on Section 8.1 and [37] with the caveat that summary arcs must be computed for external functions[6]. These summary arcs can be placed conservatively (i.e., from every input to every output, labelled with an asterisk constraint), but we recommend (and have implemented) custom summary arcs for common functions, such as `lists:nth/2` in Erlang[7].

Interprocedural arcs are labelled with the empty constraint ($\varnothing$), as they do not affect the traversal, and just serve to connect calls and arguments to functions and parameters.

The slicing algorithm incorporates both the traversal restrictions of the CE-PDG and the table to build summaries on-the-fly from tabular slicing. Thus, a table is created, to store the current and the contextual *state* of the slice. These states match the current and contextual node of tabular slicing, but will store more information, as the CE-PDG part of the algorithm needs to remember the state of the stack and the type of the last arc traversed. Thus, the current state contains a node reached during traversal, the stack with which it was reached, and the type of the arc used to reach it. The contextual state only contains the formal-out node that acts as context and the stack that reached it.

[6] External functions are those that are not represented in the graph, such as the language's API, external libraries or even part of the source code that has been excluded from the analysis to speed it up.

[7] This function takes an index i and a list as arguments, and outputs the i-th element of the list. In order to enable precise slicing of lists in the presence of this method, we used an access constraint $\rceil_i$, so that the behaviour of the function is readily available to the slicer. In cases when the index cannot be computed from the source code (specifically in static slicing), a star index is used ($\rceil_*$), which matches any index.

8.3.1 The slicing algorithm

The algorithm traverses the graph normally, using the current state to apply the label constraints and other traversal limitations from Algorithm 7, and only taking into account the contextual state when reaching interprocedural nodes: actual-out, formal-in, and *Enter*.

Algorithm 9 describes the process required to produce a slice that is context- and field-sensitive. The main function is SLICE (lines 1-20), and the process starts by creating three global sets that will store the states that have been seen (*pathEdge*, line 2), the states that have not been processed yet (*workList*, line 3) and the summary arcs that have been found (*summaries*, line 4). The first two contain 5-tuples which contain the current state (current node, current stack and last arc's type) and the contextual state (contextual node and its stack). The only relevant type of arc is the structural arc, so, to lower the amount of possible entries, we only allow two values: structural arcs (*str*) and any other ($\bullet$). As with the tabular algorithm, the contextual node might not be present ($\circ$). The first work item is the slicing criterion (line 5), with an empty stack ($\perp$), no previous arc ($\bullet$) and no context ($\circ, \perp$). Then, the main loop of the algorithm (lines 6-19) processes each work item, until the *workList* is empty. Finally, the current node of each element in *pathEdge* is extracted (line 20), obtaining the slice as a set of nodes.

Each work item is processed following tabular slicing's pattern: *Enter*, formal-in and actual-out nodes require special handling, as they are the target of interprocedural arcs (and we are considering backward slicing), and all other nodes just traverse arcs as normal. An item (containing a current state —n, s, t— and a contextual state —n', s') is extracted from the *workList* (line 7), and processed:

- If n is an *Enter* node (line 8), the traversal is intraprocedural if it has a context ($n' \neq \circ$, i.e., we "are" in Phase 2), and interprocedural otherwise (Phase 1).
- If n is an actual-out node (line 10), we traverse intraprocedural arcs exclusively. Then, we traverse all summaries found previously (lines 12-13) and we apply a context switch (lines 14-15), i.e., the formal-out that matches our actual-out[8] becomes the current and contextual node.
- If n is a formal-in node (line 16), we traverse interprocedural arcs with the same conditional as with *Enter*, and when we do have a context ($n' \neq \circ$, line 18), we can create summary arcs, as we have found a

[8]The algorithm uses a **for all**, because a given actual-out can have multiple matching formal-out nodes, such as with virtual calls in object-oriented languages and multi-clause functions in functional languages.

Algorithm 9 Tabular-constrained slicing algorithm for the CE-PDG.

Input: $G = (N, A)$ is a CE-PDG, $sc \in N$ is the slicing criterion, A is subdivided into control (A_c), flow (A_f), structural (A_{str}), input (A_{in}), call (A_{call}) and output (A_{out}), $A_{inter} = A_{in} \cup A_{call} \cup A_{out}$, $\mathcal{S}$ is the set of all possible stacks, k is the maximum size a stack can reach, $N_{in}, N_{out} \subset N$ are the actual-in/out nodes of G.

Output: A set of nodes that form the slice of G w.r.t. sc.

1: **function** SLICE($G = (N, A), sc$)
▷ The following three variables are global for the algorithm.
2: $pathEdge \leftarrow \emptyset$ ▷ Elements $\in N \times \mathcal{S} \times \{str, \bullet\} \times (N \cup \{\circ\}) \times \mathcal{S}$.
3: $workList \leftarrow \emptyset$ ▷ Elements $\in N \times \mathcal{S} \times \{str, \bullet\} \times (N \cup \{\circ\}) \times \mathcal{S}$.
4: $summaries \leftarrow \emptyset$ ▷ Contains summary arcs $\in N_{in} \times \mathcal{S} \times N_{out} \times \mathcal{S}$.
5: PROPAGATE($\langle sc, \bot, \bullet, \circ, \bot \rangle$)
6: **while** $workList \neq \emptyset$ **do**
7: $w \leftarrow$ POP($workList$) ▷ $w = \langle n, s, t, n', s' \rangle$
8: **if** ISENTER(n) **then**
9: TRAVERSEARCS($w, n' \neq \circ$)
10: **else if** ISACTUALOUT(n) **then**
11: TRAVERSEARCS($w, true$)
12: **for all** $\langle n_{in}, s_{in}, n, s \rangle \in summaries$ **do** ▷ Follow summaries.
13: PROPAGATE($\langle n_{in}, s_{in}, \bullet, n', s' \rangle$)
14: **for all** $n_{fo} \mid (n_{fo}, n) \in A_{out}$ **do** ▷ n_{fo}: formal-out node.
15: PROPAGATE($\langle n_{fo}, s, \bullet, n_{fo}, s \rangle$)
16: **else if** ISFORMALIN(n) **then**
17: TRAVERSEARCS($w, n' \neq \circ$)
18: **if** $n' \neq \circ$ **then** CREATESUMMARIES(n, s, n', s')
19: **else** TRAVERSEARCS($w, true$)
20: **return** $\{n \mid \langle n, s, t, n', s' \rangle \in pathEdge\}$
21: **function** PROPAGATE($e = \langle n, s, t, n', s' \rangle$)
22: **if** $e \notin pathEdge$ **then**
23: $pathEdge \leftarrow pathEdge \cup \{e\}$
24: $workList \leftarrow workList \cup \{e\}$
25: **function** TRAVERSEARCS($w = \langle n, s, t, n', s' \rangle, intraOnly$)
26: **for all** $(n'', n) \in A$ **do**
27: **if** $\neg(intraOnly \wedge (n'', n) \in A_{inter}) \wedge \neg(t = str \wedge (n'', n) \in A_f)$ **then**
28: **for all** $s'' \in$ RESOLVECONSTRAINT($s, (n'', n)$) **do**
29: **if** $(n'', n) \in A_{str}$ **then** $t' \leftarrow str$
30: **else** $t' \leftarrow \bullet$
31: PROPAGATE($\langle n'', s'', t', n', s' \rangle$)

```
32: function CREATESUMMARIES(n_fi, s_fi, n_fo, s_fo)      ▷ fi and fo: formal-in/out.
33:     for all ⟨n_in, n_out⟩ ∈ ACTUALNODES(n_fi, n_fo) do
34:         if ⟨n_in, s_fi, n_out, s_fo⟩ ∉ summaries then
35:             for all ⟨n_out, s_fo, _, n, s⟩ ∈ pathEdge do      ▷ Visited actual-outs.
36:                 PROPAGATE(n_in, s_fi, •, n, s)      ▷ Apply the new summary.
37:             summaries ← summaries ∪ {⟨n_in, s_fi, n_out, s_fo⟩}
38: function RESOLVECONSTRAINT(s, e)      ▷ Apply Table 8.1.
39:     c ← ARCCONSTRAINT(e)
40:     ⟨op, pos⟩ ← c
41:     if c = AsteriskConstraint then return {⊥}      ▷ Case 5
42:     else if c = EmptyConstraint then return {s}      ▷ Case 1
43:     else if op = { ∨ op = [ then PUSHLIMITED(c, s, k)      ▷ Case 2
44:     else if s = ⊥ then return {⊥}      ▷ Case 3
45:     else if op = } ∧ TOP(s) = ⟨{, pos⟩ then POP(s)      ▷ Case 4a
46:     else if op = ] ∧ TOP(s) = ⟨[, pos⟩ then POP(s)      ▷ Case 4b
47:     else return ∅      ▷ Error, no case has been matched.
48:     return {s}
```

connection between a formal-out (our context) and a formal-in node. The creation of summary arcs is described below.

- Otherwise, we traverse intraprocedural arcs (lines 19).

To traverse arcs, we call the aptly named TRAVERSEARCS function (lines 25-31). It takes two arguments, a work item whose incoming arcs should be traversed and a boolean *intraOnly*, which controls whether interprocedural arcs should be ignored. It extracts all arcs whose target is the current node n (line 26). It then applies a filter (line 27), continuing only through intraprocedural arcs (if *intraOnly* is *true*, left side of the $\wedge$), and excluding flow arcs if the previous arc was structural (right side of the $\wedge$). The first filter comes from tabular slicing, and the second, from the CE-PDG. Once verified that the arc can be traversed, it resolves the arc's constraint with the call to RESOLVECONSTRAINT (line 28). This function applies the logic from Table 8.1[9] (lines 38-48), and returns a set with one element (the new stack) or an empty set (the arc cannot be traversed). In the former case, the type of the arc (t') is simplified to any of *str* or $\bullet$ (lines 29-30), and the new work item is added to the *workList* (line 31).

The *summaries* set contains arcs that connect an actual-in node (and its stack) to an actual-out node (and its stack), resulting in a 4-tuple. It is used to

[9]Case 2 is slightly modified, compared to the constrained algorithm (Algorithm 7), using PUSHLIMITED instead of PUSH. This is k-limiting, which is further explained on Section 8.3.2.

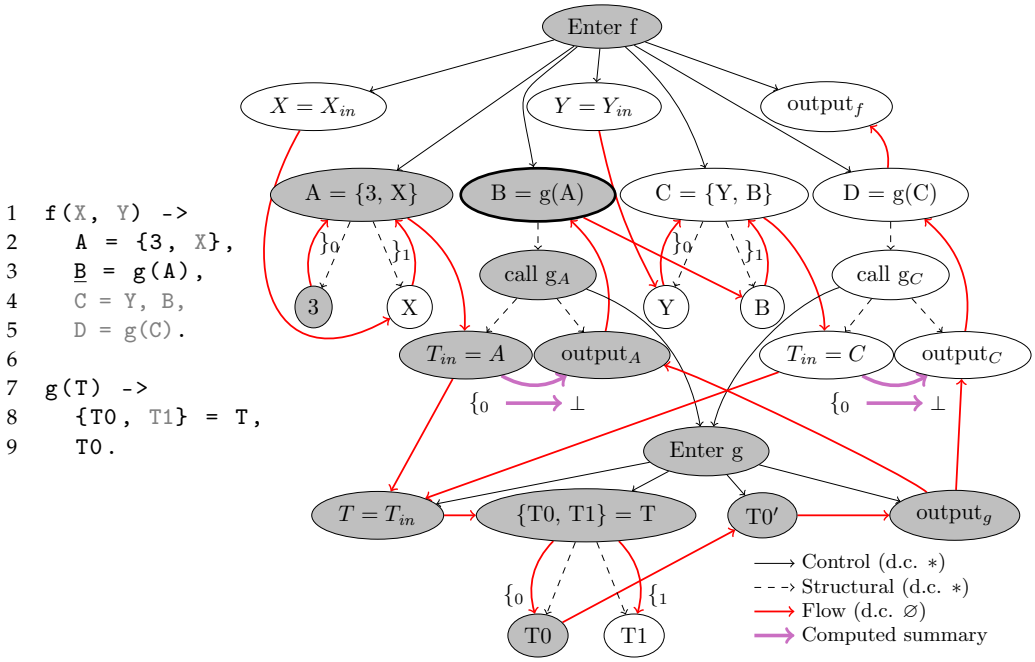


FIGURE 8.6: Slice of Fig. 8.2 w.r.t. ⟨3, B⟩ (Algorithm 9).

store summaries as they are found, and its contents are read in the main SLICE function (when an actual-out node is found). New summary arcs are created when a formal-in is reached, by calling the CREATESUMMARIES function. This function takes four parameters: a formal-in, its stack, a formal-out, and its stack. The function (lines 32-37) first obtains the pairs of actual-in/out nodes that match the given formal-in/out (line 33). It the actual-in/out pair, along with the stacks passed as arguments, is not present in *summaries* yet (line 34), two actions are performed: apply the new summary arc retroactively and store it in *summaries*. The latter is performed on line 37. For the former, all actual-out nodes that have been reached already must take this new summary arc into account. Thus, a lookup for matching work items in *pathEdge* obtains a list of candidates (line 35), and they are added to the *workList* (line 36).

Example 8.6 (Slicing Fig. 8.2 with Algorithm 9)**.**
If we apply the tabular-constrained slicing algorithm, we would obtain the slice shown on Fig. 8.6. We can see that two nodes ("X") and "X = X_{in}") have been removed, reaching the minimal slice. Summary arcs are also shown here: one has been computed and placed at all calls to g (although only one of them has been traversed). This time, summary arcs are not just a connection between two nodes, but between two nodes and their stacks.

```
1  f(X, Y) ->
2    A = 3, X,
3    B = g(A),
4    C = {Y, B},
5    D = g(C).
6
7  g(T) ->
8    {T0, T1} = T,
9    T0.
```

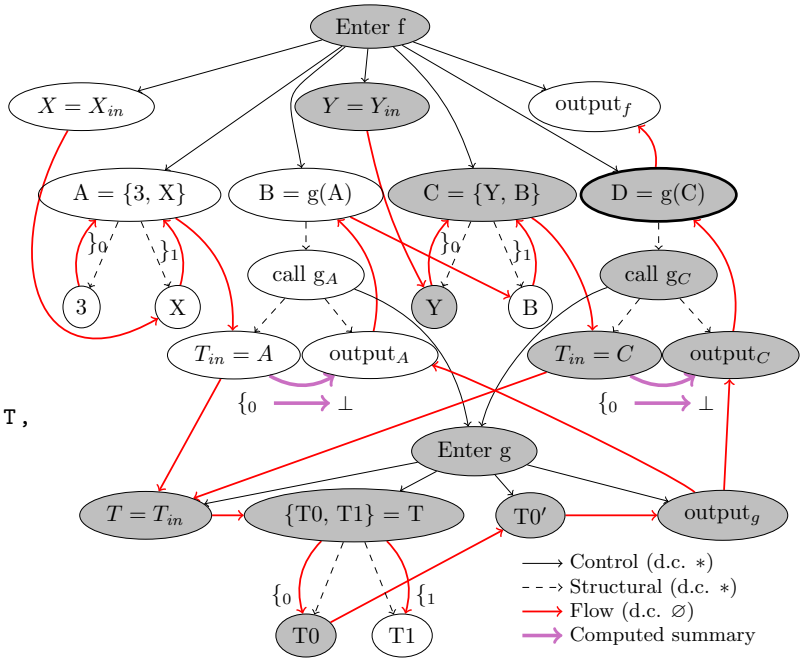


FIGURE 8.7: Slice of Fig. 8.2 w.r.t. $\langle 5, \texttt{D} \rangle$ (Algorithm 9).

Thus, from any of the actual-out nodes and an empty stack, the effect of traversing g is that the actual-in is reached with $\{_0$ as the stack. Otherwise, the execution of the algorithm is a union of the two algorithms that make it up.

Revisiting the original PDG and the slice extracted from it, we can see the context- and field-sensitive slice in Fig. 8.7. Thanks to the tabular and constraint sides of the algorithm, it avoids including the assignments to `A` and `B`: interprocedural arcs and data structures are traversed only when necessary, producing the minimal slice w.r.t. $\langle 5, \texttt{D} \rangle$.

8.3.2 Loops and stack limits

In the CE-PDG, special care was taken to avoid an infinitely growing stack due to flow dependence loops. It introduced stack ordering, loop detection and neutralisation, etc. However, in the tabular-constrained algorithm, we found out, empirically, that *k*-limiting was much faster than those measures, while retaining the same amount of precision (if a high enough *k* was chosen). For this reason, in this chapter's algorithms, there is no mention of these countermeasures, and the resulting algorithm uses *k*-limiting to avoid becoming trapped on an infinite loop.

We applied k-limiting by simply replacing the PUSH operation on case 2 with a PUSHLIMITED operation, which removes the constraint at the bottom of the stack when the given limit (k) is reached.

8.3.3 Asymptotic temporal cost

This algorithm combines the costs of the tabular and constrained slicing algorithms. In the classic tabular algorithm, the table contained at most $n \log n$ elements, where n is the number of nodes in the graph. The CE-PDG introduces new nodes and adds three elements: two stacks and the last arc's type. Each stack has $s^k + 1$ possible values, where s is the number of different open constraints that appear in the graph, and k is the maximum depth of the stack (a user setting in the slicer). The last arc's type has two possible values. Altogether, the table can contain up to $2(s^k + 1)^2 n \log n$ elements. Thus, the temporal efficiency of the tabular-constrained slicing algorithm is $O(s^k n \log n)$.

8.4 Empirical evaluation

The algorithms described throughout the chapter have been implemented in a slicer for Erlang, called *eKnife*. More details on its implementation, and instructions to install it or try the demo are available on Chapter 10. We have also implemented the standard PDG slicing algorithm (1), such that we can ask the following questions:

RQ1 What is the cost of introducing each sensitivity into the algorithm?
RQ2 What is the precision gained by introducing each sensitivity?
RQ3 Which of context-sensitivity and field-sensitivity is more powerful?

To answer these questions, we evaluated our algorithms with Bencher[10], a benchmark suite of Erlang programs built for program slicing, which contains the minimal slice for each program (the gold standard). The evaluation was performed on an Intel Xeon E-2136 CPU running Debian Linux 11, with 32GB of DDR4 RAM available. The benchmark process was given a higher priority, and non-critical processes were stopped during the benchmark. A CE-PDG was built for each benchmark, with the required summary arcs for external functions. Then, multiple slices were extracted from each graph, using as slicing criteria all variables in functions with any composite data structure.

For each slicing criterion, four different slicing algorithms: the classic PDG (Algorithm 1), constrained (Algorithm 7), tabular (Algorithm 8) and tabular-constrained (Algorithm 9) slicing algorithms. Each slice was produced 1001

[10] Available at `https://mist.dsic.upv.es/bencher/`.

TABLE 8.3: Absolute slice size and time to produce slices with different algorithms.

Algorithm	Size	Time
PDG	235.36 n.	0.28 ms
Tabular	231.01 n.	3.25 ms
Constrained	192.40 n.	2.23 ms
Tab.-Constr.	184.20 n.	55.36 ms

TABLE 8.4: Relative slice size and time to produce a slice between different algorithms.

Algorithms compared	Precision increase	Time increase
Constrained *vs.* PDG	9.70 %	5.66
Tab.-Constr. *vs.* Tabular	10.41 %	8.21
Tabular *vs.* PDG	1.40 %	8.18
Tab.-Constr. *vs.* Constrained	2.31 %	11.65
Tab.-Constr. *vs.* PDG	11.80 %	97.18

times, discarding the first execution to avoid the influence of caching in the results. In total, 729 slicing criteria have been sliced 1000 times with each of the four algorithms, totalling just over 2.9 million slices.

Our analysis has produced results with absolute (Table 8.3) and relative (Table 8.4) averages regarding the size of the slices produced (in number of nodes), and the time required to produce a single slice. The absolute values show that both the constrained and tabular algorithms produce a ten-fold increase in the time required to produce slices, and reducing the average size of a slice. However, because the variance of these averages is very high, we prefer to focus on the relative results.

The relative results have been computed by comparing the time and size of the same slicing criterion, when sliced with two algorithms. We first obtain the ratio between the time (or size) taken by the algorithm, and then we average all the ratios to produce the numbers shown in Table 8.4. There, we can see that switching from the standard to the constrained algorithm makes it 5.66 times slower. We can see that the largest effects, precision-wise, are when constraints are added (i.e., field sensitivity, in the 1st, 2nd and last row), achieving an average 10 % increase. The addition of tabular slicing (i.e., context sensitivity, in the 3rd and 4th row) is not as relevant, with improvements below 2.5 %. This table answers RQ2 and RQ1, though the latter is not as clear. Adding

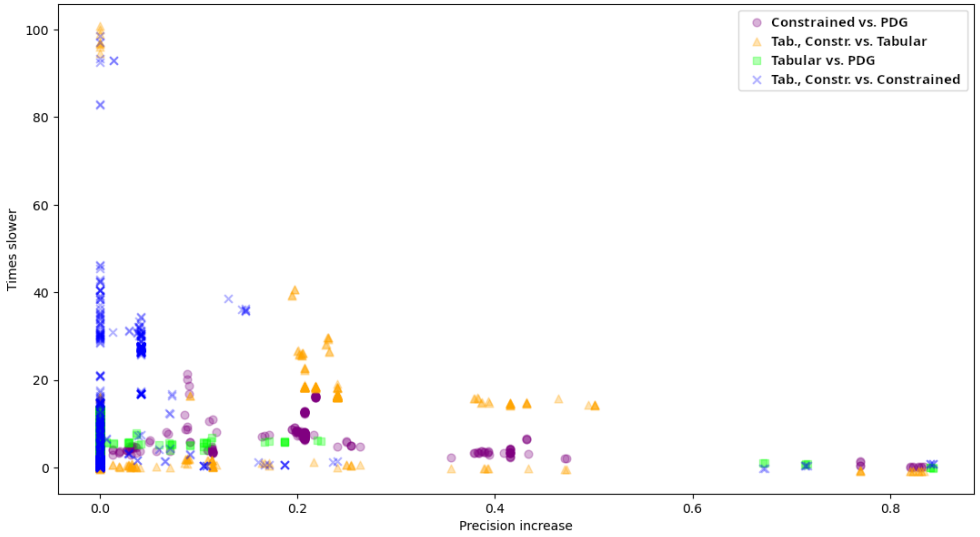


FIGURE 8.8: Comparison of the precision increase against the relative time required of adding a single sensitivity (context or field).

field-sensitivity (first two rows) incurs a cost of 5-8×, while adding context-sensitivity (with the tabular algorithm) has a higher cost (8-11×). Combining both results in an average of 100× slowdown.

Figure 8.8 plots the precision gained vs. the time lost for every single slicing criterion, comparing four different pairs of algorithms. In each pair, the algorithm becomes either context-sensitive or field-sensitive. We can see that there are a few data-points in the lower-right corner, which means that they have an increased precision of 70-80 % with a very small time penalty. On the opposite side of the spectrum, the ones in the upper-left corner have no precision increase, but its traversal takes 100 times longer. If we analyse each algorithm comparison, we see that *Tab., Constr. vs. Constrained* is mostly located along $x = 0$ (no precision gains, variable time), whereas the rest form horizontal lines: *Tabular vs. PDG* could be characterised as $y = 5$ (roughly 5 times slower, varying precision), and *Constrained vs. PDG* as $y = 3$. All of them have outliers throughout the graph, but the highest variance is found in *Tab., Constr. vs. Tabular*, which could be described as $y = 0$, but also has elements around $(0.2, 20)$ and $(0.4, 20)$.

All in all, the effect of adding a given sensitivity is hard to generalise across a diverse set of benchmarks. Thus, RQ3 cannot be answered without a benchmark that is representative of the desired application. Therefore, each algorithm can be significant in a different context, as the simpler ones can

speed up the process, while more sensitive ones might produce the smallest results. For some applications, speed might be more relevant, while others might take advantage of a more precise slice.

8.5 Related work

The closest approach to our own is Perez's PhD thesis [88, Chapter 4.4], which extends the CE-PDG to place summary arcs during the graph's construction. To this end, he utilises a set of grammars that summarise the constraints that one would find by traversing the body of a function, applying those grammars to attempt to traverse the summary arc. However, the resulting arcs cannot correctly represent cycles in the graph, either from imperative loops or from recursive calls.

Of course, the SDG model is not the only context-sensitive alternative in slicing, just the most efficient. Sharir and Pnueli's approach [98] uses a stack similar to the CE-PDG's to keep track of the calls entered and exited. This approach has been used for techniques that are incompatible with the SDG, such as Krinke's threaded Interprocedural PDG [59]. This technique, just as the constrained algorithm, introduces an exponential element in the asymptotic cost.

Notable alternative approaches to field-sensitive slicing include *atomization* [7, 46, 58], which splits data structures into simpler assignments, such that each instruction affects one memory location. This is a valid approach for non-recursive data structures, but fails in their presence, as it must represent the sub-element of the structure being (un)packed. In contrast, the CE-PDG only unfolds the elements that appear in the source code, so it can represent recursive structures.

Part IV

Tools & Software

Chapter 9

JavaSlicer

JavaSlicer is a program slicer for Java written in Java. It is based on the SDG and implements the new techniques and improvements detailed on Chapters 3 and 4, and others from previous works [34]. It is distributed as free software, under the Affero GNU Public License 3.0 (AGPL), and provides building blocks to quickly try modifications to the SDG and its variants.

9.1 Usage

JavaSlicer contains a library usable by other developers and a simple console command that can easily produce slices with minimal user effort.

To use it, the user needs to have Java 11 or later installed (and Maven if they wish to compile the software), and needs to download the source code from its repository (or its mirror on GitHub):[1]

```
https://mist.dsic.upv.es/git/program-slicing/SDG
https://github.com/mistUPV/JavaSlicer
```

Then, the software must be compiled with Maven:

```
mvn package -Dmaven.test.skip
mv sdg-cli/target/sdg-cli-*-with-dependencies.jar sdg-cli.jar
```

Alternatively, the user can download a pre-built release from GitHub's releases page[2]. Whichever path the user takes, they should have a `.jar` file containing the compiled code, which can be executed as:

```
java -jar sdg-cli.jar [options]
```

[1] At time of writing, the latest commit was `542b39d8`. *JavaSlicer* may be developed further, and this chapter may no longer be accurate for future versions.

[2] `https://github.com/mistupv/JavaSlicer/releases`. Currently, the latest release is v1.3.1.

JavaSlicer requires a few options to indicate the program to be sliced, the desired location for the slice and the slicing criterion, which can be specified with the following flags:

- `-f,--file [Java file]` The Java file to be sliced.
- `-l,--line [number]` The slicing criterion's line number.

These two options can be shortened to `-c,-criterion [file#line]`, where the argument is the slicing criterion expressed as Java file and the line number, separated by `#`. Additionally, the behaviour of the slicer can be tweaked with the following options:

- `-v,--var [name]` To indicate the variable of the slicing criterion. The variable can also be indicated on the criterion option (`-c`) by adding `:name` as a suffix, such that the argument becomes `file#line:variable`.
- `-o,--output [path]` The directory where the resulting slice should be placed. E.g., if the Java file is called `Example.java`, the slice will appear at a file with the same name in the path indicated. This option defaults to `./slice/`.
- `-i,--include [path]` For cases when the program to be sliced spans across multiple files (which will be most Java programs for which slicing is relevant), this option loads additional source code from the folder indicated (it must be the root of a series of Java packages) and populates the graph with it. Example 9.2 expands on this topic.
- `-t,--type [SDG|ASDG|PSDG|ESSDG|JSysDG]` Selects the type of SDG that will be used to produce the slice. By default, the JSysDG is used. Each graph includes the cumulative improvements from previous graphs. Selecting a simpler graph can make the output of the slicer less precise and incomplete (i.e., the output is not a slice), but can greatly improve the slicer's performance.

  SDG The basic system dependence graph for context-sensitive slicing of simple assignments, loops and `if`-statements, with primitive variables.

  ASDG The augmented system dependence graph, adds support for unconditional jumps (e.g., `break`, `return`... see Chapter 4).

  PSDG The pseudo-predicate system dependence graph, improves precision compared to the ASDG when multiple unconditional jumps are present (see Chapter 4).

  ESSDG The exception-sensitive system dependence graph, adds support for `throw` and `try-catch` (see Chapter 3).

  JSysDG The Java system dependence graph (first defined by [105] and later refined by [34]) includes support for the object-oriented side of Java programs (objects, polymorphism, inheritance...).

- `-h,--help` Shows all the options with some help text. This can be useful when new options are added to *JavaSlicer*, as otherwise this chapter contains a more detail account than what this flag provides.

Given that most options are optional, simple slices can be easily extracted:

Example 9.1 (Simple *JavaSlicer* usage)**.**
Given a Java program stored at *Example.java*, to produce a slice w.r.t. $\langle 10, \texttt{x} \rangle$ would require the following command:

```
java -jar sdg-cli.jar -c Example.java#10:x
```

When the execution ends, a new folder named `slice` will be created, and within it, the slice will be in a file called `Example.java`.

However, most programs span across multiple files, and thus require the use of `-i` to tell *JavaSlicer* where the Java sources are:

Example 9.2 (Using *JavaSlicer* on a multi-file program)**.**
A developer needs to debug an issue with their code, located in a folder called `src/`. One method (`Triangle#area()`) is returning a wrong value, and they want to trace the source of this error, so they include the folder and select `Triangle.java` as their slicing criterion, adding the corresponding line an variable, ending with the following call to *JavaSlicer*:

```
java -jar sdg-cli.jar -i src -c src/Triangle.java#86:res
```

In this case, a much larger SDG is generated, and the output folder (`./slice` by default) will contain a sliced `.java` file for each source file that was included in the slice.

However, when the source code of part of the program, such as third party libraries, is not available (or not relevant to the slice), they must be loaded to Java's classpath, or otherwise there will be undefined method calls, which will cause errors. These classes will not be part of the SDG, but will be treated as calls to the Java library. To do this, add the argument `-cp [CP]` between `java` and `-jar`, where `[CP]` is a sequence of `.jar` files separated by semicolons, for example:

```
java -cp ui.jar:math.jar -jar sdg-cli.jar -c Test.java#10:x
```

9.1.1 Alternatives: Docker and demo

Users that are unable to run a `.jar` with Java locally can also try *JavaSlicer* online, through a limited demo, accessible at `https://mist.dsic.upv.es/JavaSlicer/demo`. In this webpage, the user can write or load a single Java file, select a slicing criterion and slice it with the JSysDG. This demo has fewer options, and it also has time and memory limits to avoid abuse, but makes the slicer accessible to users unfamiliar with terminal commands.

Another option for users unable to install Java, there is an option to use *JavaSlicer* from a Docker container. The `Dockerfile` included in the repository can be used to build a Docker image, which will download dependencies and compile the slicer with Maven. To build the image, run the following from the main folder of *JavaSlicer*:

```
docker build --tag javaslicer
```

Then, the image can be run as a container, mounting the `examples` folder as a volume to share files between the container and the host:

```
docker run -i --tty --rm --volume=./examples:/src javaslicer
```

After running this command, a new shell opens, running Ubuntu 20.04, with *JavaSlicer* ready to run. Running the `exit` command will exit the container, deleting all changes made outside of `/src`. Finally, to delete the image, you can simply run `docker rmi javaslicer`.

Example 9.3 (Running *JavaSlicer* in a Docker container).
Inside the container, *JavaSlicer* is installed to `/SDG/javaSDG.jar`, so the following command slices `Example1.java` (mounted as a volume) w.r.t. $\langle 11, \texttt{sum} \rangle$:

```
java -jar /SDG/javaSDG.jar -c /src/Example1.java#11:sum \
    -o /src/out
```

The output will be placed at `/src/out`, and will also appear in the host system, so that the user can retrieve it and store it permanently.

9.1.2 Use cases

Ariel is a third year Computer Engineering student, and has been tasked with completing a large Java project. He has used some software patterns, but multiple processes are mixed throughout the code. To separate them, he uses *JavaSlicer*'s CLI to perform multiple slices with respect to the processes' results.

With these, he determines which methods appear in both slices and should be separated.

Alex is a teaching assistant, tasked with marking student assignments. Some cases can be difficult, as students tend to leave unnecessary code near the solution as they make their way through a problem. To prune the code without altering its behaviour, they use *JavaSlicer*'s demo website, copying the single-file submissions and selecting the return value of a key method as the slicing criterion. With the slice, most unnecessary code is removed, and the remaining code can be easily understood.

Ada is debugging a program she uses. She is not familiar with the source code, but has located a variable with an illogic value. With *JavaSlicer*'s jar, she slices the whole project, removing whole swaths of code, and making her task much easier.

9.2 Implementation

JavaSlicer manages its dependencies through Maven, of which there are two main ones: *JavaParser*[3] and *JGraphT*[4]. The former not only provides parsing of Java programs, but also contains a symbol solver, which given a variable name and location, is able to identify which syntactic element defined it. The latter contains several classes that represent graphs efficiently.

The software itself is organized in three submodules, `sdg-cli` (the terminal client we described in the previous section), `sdg-bench` (for benchmarks) and `sdg-core` (the slicing library itself). The easiest way to describe the implementation is describing the process of obtaining a slice, from start to finish. We start at `sdg-cli` with:

1. Parsing *JavaParser* must be first configured to be able to recognize function calls to JRE functions and, when slicing multiple files, load all the folder to detect them as well. Each file parsed produces a `CompilationUnit`, which contains its package declaration, import statements, classes, etc. With a collection of these units, the `SDG` class or one of its subclasses (which match the types available in the previous section) is instantiated and run the build process, entering `sdg-core`.

2. The Class Graph This graph is used to help with object-oriented features such as virtual calls. It contains classes, interfaces and methods as nodes, and their relationships (contains, inherits, extends) as arcs.

[3] `https://javaparser.org/`
[4] `https://jgrapht.org/`

3. Building CFGs Each compilation unit contains multiple classes, which themselves contain constructors and methods. For each constructor and method declaration (grouped as `CallableDeclaration`), a CFG (or a subclass, depending on the type of SDG being built) will be built. This process uses the visitor pattern, traversing the abstract syntax tree starting at the aforementioned declaration. The visitor pattern is very useful for structured loops and branching statements (`if`, `for`, `while`), but requires external lists to keep track of unresolved jumps, like `break` or `continue` (both labelled and unlabelled). When each statement is processed, a list of `VariableActions` is extracted, containing elements like a variable was defined, another used, a call was made, which used and defined other variables, etc. This list will be later used to compute flow dependence, and to add call and actual nodes.

4. The Call Graph This graph is necessary for later steps, and it is a simple structure that shows the methods and calls of a program, with the former being nodes, and the latter, arcs between them.

5. Data-flow Analysis In a simpler language, this step would be unnecessary, but in Java there are both class and static fields that act as global variables, and are declared outside the block we analyse for the CFG and PDG (a single method or constructor declaration), so to find out which variables are affected (used or defined) in each method, we need to perform a data-flow analysis. We perform two analyses, both over the call graph. The first finds definitions without a matching declaration, and the second, usages with no definition or declaration. Definitions and usages found are added to the *Enter* node and call nodes of any method and call (respectively) that directly or indirectly define or use that variable.

6. Building PDGs Each CFG is transformed into a PDG: nodes are copied over to a new graph, then control and flow dependences are computed to create the corresponding arcs between nodes. Call, actual and formal nodes are added to the graph, based on steps 3 and 5. Finally, the resulting PDG (all nodes and arcs) are copied to the SDG.

7. Connect Calls Once all PDGs are merged into a single SDG, input, output and call arcs are added, connecting actual to formal nodes, formal to actual nodes and call to *Enter* nodes, respectively.

8. Compute Summary Arcs To finish the computation of the SDG, summary arcs are computed, in another data-flow analysis over the call graph, as each

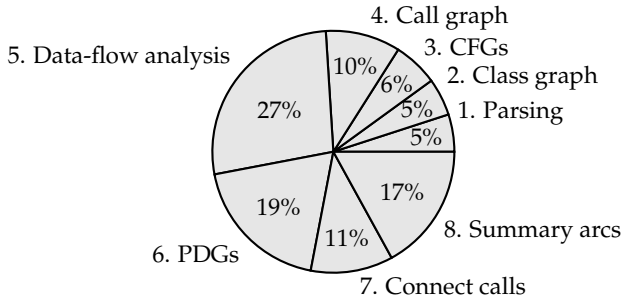


FIGURE 9.1: Breakdown of the time dedicated to each step of the creation of the SDG.

summary arc that is added may alter summary arcs in methods that call it. Once this process has finished, we can use this graph multiple times to produce slices.

9. Slice To produce a slice, we need a complete SDG, which we just built, and a slicing criterion. The library only requires a file name (to distinguish between multiple compilation units) and a line number, taking the first statement in that line as slicing criterion. With the node, we apply the algorithm that corresponds to the graph we are about to slice, and produce a set of nodes that form the slice.

10. Pruning and Printing With the set of nodes, we prune a copy of the AST (to avoid changing objects that are present in the SDG), and then use *JavaParser* to print it to a file in the given output folder.

The process' time is mostly taken up by the creation of the SDG (steps 1-8), and the breakdown is shown in Fig. 9.1. Data-flow analysis, computing dependences and summary arcs make up almost two thirds of the total time, which in the first and last show the inefficiency of having to re-compute known values until finding a stable result. In the case of dependences, there are more efficient algorithms that could have been implemented, such as Roman Chariots to compute control dependence in linear instead of quadratic time [85].

Additionally, each feature added to the slicer, and many have been added at this point (at least one per SDG type), complicates the implementation and requires it to be more inefficient to accommodate the different variations of SDG that can be produced by it.

9.3 Limitations

JavaSlicer lacks support for concurrency and threads, Java streams, and other functional features. Additionally, even though it has support for objects and related features, it lacks a pointer analysis to determine which variables may be the same (e.g., if `a == b` may be true, b.x = 10; modifies both `a` and `b`). This is because it is a research tool, built specifically to test new proposals for slicing objects, unconditional jumps, exceptions, etc., and any feature orthogonal to the techniques has been deprioritized.

Chapter 10

e-Knife Erlang

e-Knife Erlang is a program slicer for Erlang written in Java, based on the Expression Dependence Graph (EDG). It covers the sequential part of Erlang, but not its concurrent side (no process spawning or message passing). Initially, it was designed to be able to accommodate any programming language, if a parser and some semantic information was given, and there were projects that added Java and PHP to the set of supported languages that could be sliced with it. However, as the complexity of the project grew, it was limited by having to be a "jack of all trades", so the language that benefited the most from the EDG (Erlang) was kept, and others were separated to their own project (e.g., *e-Knife Java*[1]) or removed from the project.

In *e-Knife*, we have implemented the field-sensitive tabular slicing technique proposed in Chapter 8, and some elements from exception handling described in Chapter 3. It is also distributed as free software, under the AGPL 3.0.

10.1 Usage

As with *JavaParser*, *e-Knife* can be either installed locally, used via a Docker or OCI container, or tried through its online demo.

To install locally, the requirements are Java 11 or later, Make, Maven and Erlang 26. Once installed, only Java and Erlang are required, the rest can be uninstalled. The installation process starts with downloading the sources from the repository, either from the web interface or with `git`[2]:

```
git clone https://mist.dsic.upv.es/git/program-slicing/e-knife-erlang
cd e-knife-erlang
git checkout ca107d93
```

[1] Available at `https://mist.dsic.upv.es/git/program-slicing/e-knife-java`

[2] At time of writing, the latest commit was `ca107d93`. *e-Knife* may be developed further, and this chapter may no longer be accurate for future versions.

Then, with a terminal in the root folder of the project, just run `make`, and a `dist/` folder will appear, with all the necessary elements to run *e-Knife*. You can move this folder to any location, as long as the `.beam` files remain in the same position relative to the `.jar` file.

You can, of course, perform this same process in a Docker image, without needing to install any dependency and guaranteeing that installing this software will not affect the rest of your machine. Like with the previous chapter, we first build the image and then we run it, to open a shell in it.

```
docker build --tag eknife .
docker run -i --tty --rm --volume=./examples:/src eknife
```

In either the local or Docker case, you will need to locate the `e-Knife.jar` file, and you can run it as any other `.jar` file:

```
java -jar path/to/e-knife.jar [options]
```

The possible options for *e-Knife* are listed below:

- `-i,--input [path]` The file or folder where the code to be sliced is. If it is a folder, the slicing criterion's file must be indicated with `-f`.
- `-o,--output [path]` The file or folder (matching `-i`) where the slices must be placed.
- `-f,--file [path]` The file that contains the slicing criterion, relative to `-i`.
- `-l,--line [number]` The line that contains the slicing criterion.
- `-v,--var [name]` The name of the slicing criterion's variable.
- `-n,--occurrence [number]` If the given variable appears multiple times in the given line, the occurrence that must be selected. This parameter is optional, and the first occurrence (`-n 1`) is selected by default.
- `-G,--print-graph [path]` Indicates that the graph must be outputted to the file in the given path. The slice is also drawn in the graph, highlighting nodes in it in bright green, and the slicing criterion with a bold outline. It will be printed as PDF if the file ends in `.pdf`, or as a DOT graph specification (to be viewed with tools like XDot or exported to other formats) if it ends in `.dot`. Other extensions are unsupported and will cause an error. For non-trivial programs, we recommend exporting to DOT and using a viewer to highlight individual arcs, as the amount and density of arcs can make the PDF unsustainably large and messy.
- `--ignore-constraints` Uses a field-insensitive algorithm when slicing, so that, even though the graph contains constrained edges, they have no effect on the slice, which is field-insensitive. This produces slices faster, but with lowered precision (in most cases).

- `--ignore-calling-context` Uses a context-insensitive algorithm when slicing, such that slices are produced faster, but are no longer context-sensitive. Can be combined with the previous argument to produce field- and context-insensitive slices.
- `--single-function` Skips the creation of inter-procedural arcs, such as call, input, and output arcs. Then, the slices produced will not cross function bounds.
- `--help` Prints a summary of these options.

Producing slices only requires setting the input, output, and slicing criterion.

Example 10.1 (Simple *e-Knife* usage)**.**
If the user has an Erlang program stored in `example.erl`, the following command will produce a slice to be saved at `slice.erl`, with the slicing criterion $\langle 5, \texttt{A} \rangle$:

```
java -jar e-knife.jar -i example.erl -o slice.erl -l 5 -v A
```

If we also want to save the graphical representation of the graph to a PDF file, we can add the corresponding flag:

```
java -jar e-knife.jar -i example.erl -o slice.erl -l 5 -v A \
    -G graph.pdf
```

Then, again, most interesting Erlang programs span more than one file, so we can expand the previous example for Erlang projects.

Example 10.2 (Slicing multi-module Erlang programs)**.**
If we have an Erlang project whose sources are stored under `src/`, and we are interested in a math module `src/math.erl`, specifically in the slicing criterion $\langle 36, \texttt{Sum} \rangle$, we could run:

```
java -jar e-knife.jar -i src/ -o slice/ -f math.erl \
    -l 36 -v Sum
```

The output would appear in the `slice/` folder, where we would find the sliced modules w.r.t. the given criterion. We could also output the graph, but it is probably large enough to crash most PDF and DOT viewers.

10.1.1 The online demo

A limited demo is available at `https://mist.dsic.upv.es/e-knife-erlang`, where users can load their own files or an example and produce slices. This demo is also limited in time and memory, but enables the users to easily customize the graph they want to output, because it can selectively show or hide arcs by type (for example, hide control dependence). Users who want to produce field-insensitive slices can also use the constrained-edges version of the demo, with a toggle that lets user enable or disable constraints, and similar controls over graph output. It is available at `https://mist.dsic.upv.es/e-knife-constrained`.

10.1.2 Erlang versions other than OTP 26

It is very easy to use a different version of Erlang. The only requirement is that the version of *jinterface*, a library that enables communication between Java and Erlang processes must be compatible with the version of Erlang used[3]. You can indicate the desired version of *jinterface* when compiling the software:

```
make MVN_OPTIONS="-Djinterface.version=VERSION"
```

Where `VERSION` is the selected version, for example, 1.10.1 for Erlang 22.2 (shipped with Ubuntu 20.04) or 1.13.1 for Erlang 25.2 (Debian 12). If your version of Erlang does not appear on the compatibility table, the `README` file of *e-Knife* has further instructions.

10.1.3 Use cases

Jack is a junior developer, and has been assigned to an Erlang project with a long history. It uses some complex expressions, where the flow of information or the order of execution is not clear. Using *eKnife*'s graph generation feature, he creates a `.dot` file, which can be viewed interactively with *XDot.py*[4]. With the graph, the relationships become clearer and easier to understand.

Victoria is an Erlang programmer, and has written a simple server pattern (a loop that receives and processes messages) with multiple actions possible. She wants to split this program into multiple services, such that they can answer concurrently. Using *eKnife*'s executable, she produces one slice per service, and then adapts the resulting code to meet her expectations.

Christian has recently been introduced to version control. Thus, his commits are larger and more complex than is necessary. A colleague suggests the

[3] A table showing compatible versions is available at `https://mist.dsic.upv.es/git/program-slicing/erlang-jinterface`

[4] Available at `https://github.com/jrfonseca/xdot.py`

use of *eKnife* on their Erlang project. Before making a commit, Christian performs multiple slices centred on the chunks of code that are to be committed. With the slices, he can see which changes are completely unrelated and can be split into separate commits. Thanks to this guidance, he can manually disentangle the remaining chunks that are unrelated but share variables, producing small, understandable and reviewable commits.

10.2 Implementation

e-Knife is based on the EDG, so it follows a very different program-to-slice process that the one in *JavaSlicer*. The EDG is built upon the AST of a program, and the logic required to build it can be divided into language-dependent and language-independent elements.

e-Knife separates its logic in two Maven modules: `e-Knife` and `EDG`. The first contains language-dependent elements, such as the parsing and printing logic, and also logic for adding control-flow to an AST and for pruning an AST given a set of nodes that represents a slice. The latter holds all language-independent algorithms, such as control and flow dependence generation, summary arc calculation, slicing algorithms, etc.

Here are the steps followed in the slicing pipeline:

1. Parsing (`e-Knife`) An Erlang process is used for parsing, as Erlang's standard library contains a good parser that outputs the AST as a nested data structure formed by lists and tuples. The AST is then translated to Java objects through a visitor pattern. During this translation, some nodes are marked as expressions (they produce a value that can be used in other nodes). The Erlang process is shutdown, as it will not be used again until slices must be printed.

2. Labelling the AST (`e-Knife`) Control-flow arcs and value arcs are added to the AST, creating the labelled AST or L-AST. Control-flow arcs form CFG-like structures on top of the AST for each function, and value arcs represent intra-statement data dependence, e.g. `3 + C` depends on the value of `3` and `C`.

3. Split expression nodes (`EDG`) Nodes that have been marked as expressions in the first step are now split in two, leaving the original node as the source/target of control dependence, and the new node, labelled *result* is the source/target of all data dependences (value and flow). Existing control-flow and value arcs are adapted to include the new *result* nodes in their path. This transforms the graph into the L-AST, or TL-AST.

4. Compute dependences (`EDG`) Flow and control dependences are computed based on control-flow arcs. Call, input, and output arcs link all the functions of the graph, and summaries are computed and added to the graph. As described in Chapter 8, when the slices are field- and context-sensitive (which is the default), summary arcs for functions in the graph are computed on-the-fly during slicing, so in this step, only external summary arcs are placed, to connect the output of each library call to its inputs. The EDG is now built and can be sliced.

5. Slice (`EDG`) The criterion was specified very precisely (file, line, variable, occurrence), so locating it is easy: find the given occurrence of the variable in the line and file. The slicing algorithms varies greatly depending on which features are enabled or disabled by command-line arguments, and can range from a simple reachability algorithm to a two-pass traversal or a tabular algorithm with a stack to hold constraints. This step results in a set of nodes from the EDG.

6. Printing the slice (`e-Knife`) A clone of the EDG is made, and the copy is pruned according to the set of nodes. Then the EDG is transformed back to an Erlang AST, a series of nested lists and tuples. An Erlang process is started to print the code, which is written to a file.

As with most slicing graphs, the graph itself can be reused, producing multiple slices without repeating steps 1-4 (as long as the program being sliced is the same). *e-Knife* implements most variations of slicing as different slicing algorithms, of which the most relevant ones are:

StandardAlgorithm Context-sensitive (two-pass like the SDG), field-insensitive.
OnePassStandardAlgorithm Context- and field-insensitive.
OnePassConstrainedAlgorithm Context-insensitive, field-sensitive.
TabularAlgorithm Context-sensitive (tabular slicing), field-insensitive.
ConstrainedTabularAlgorithm Context- and field-sensitive (tabular).

As described in Chapter 8, no context- and field-sensitive algorithm with the two-pass traversal exists, as precomputed summary arcs cannot represent constraints properly, and those summary arcs are required for the two-pass traversal algorithm.

Chapter 11

reverCSP

> People assume that time is a strict progression of cause to effect, but actually [...] it's more like a big ball of wibbly-wobbly, timey-wimey stuff.
>
> *"Blink", Doctor Who (2007)*

reverCSP is a causal-consistent reversible debugger for CSP, written in Erlang. It implements in Erlang the theory described in Chapter 5. The user can interact with it through a terminal user interface, and obtain the resulting R-track as a PDF.

11.1 Usage

reverCSP has been published as free software on GitHub, under the AGPL 3.0. It can be installed locally on Linux systems, or run through a OCI/Docker container on any operating system. In any case, the source code must be first downloaded with:

```
git clone --recursive https://github.com/tamarit/reverCSP
cd reverCSP
```

Once inside the software's root folder, the user can build the container image, for example, with Docker:

```
docker build --tag reverCSP .
```

Otherwise, the software can be built locally (only in Linux systems). The requirements to build *reverCSP* are *make*, *git*, and *erlc* (the Erlang compiler). The software can be built with:

```
make
```

```
Current expression:
(P [|{|a|}|] R)

These are the available options:
1 .- P
2 .- R
3 .- Random choice.
4 .- Random forward-reverse choice.
5 .- See current trace.
6 .- Print current R-track.
7 .- Reverse evaluation.
8 .- Undo.
9 .- Roll back.
0 .- Finish evaluation.
What do you want to do?
[1/2/3/4/5/6/7/8/9/0]:
```

FIGURE 11.1: Main menu of *ReverCSP*

Now the software can be executed. If run locally, Erlang 22 (or later) and Graphviz must be installed. When run inside a container, it must be started, using two volumes to map the input (the *examples* folder) and the output:

```
docker run -it -v $PWD/examples:/reverCSP/examples \
        -v $PWD/output:/reverCSP/output --rm reverCSP
```

In either case, the first step is running the binary, with a single parameter: the CSP specification to be debugged. For example:

```
./reverCSP examples/ex1.csp
```

The execution of *reverCSP* features two main phases:

R-tracks generation. R-track generation can be done with a random execution or with a user-directed execution. Thus, the user can choose at any step the applied rule. The R-track is generated dynamically. This means that the user can perform, say, 100 random steps, then go backward, say 10 steps, and then go forward again selecting a different rule to be applied. Thus, a new R-track is dynamically generated for each new forward step performed.

R-tracks exploration. The R-track can be traversed forward or backward with either the deterministic or the causal-consistent semantics.

These two phases are interleaved according to the users' actions, transparently choosing the appropriate one depending on the current R-track's state.

At any step, we can see the current expression, the trace, and the R-track (they can be printed as PDF).

When *reverCSP* is run, it loads the CSP specification into memory, and then presents a menu like the one shown on Fig. 11.1. The options available to the user are:

Perform a single step In options 1 and 2, the user can select which step of the process (`P` or `R`) is executed next. Option 3 picks a random step among the available ones. Option 4 picks a random option, while including reverse evaluation steps in the random draw. These options only appear if there are available steps to be taken in the current direction (forwards by default).

Current state Options 5 and 6 show to the user, respectively, the current trace (the steps taken that have not been reversed) and the current R-track. The former is shown on the terminal, while the latter is printed to a PDF file, with a style similar to the one used on Chapter 5. The track is printed using Graphviz[1], so the graph specification file (`.dot`) is also available to the user.

Reverse evaluation Option 7 switches from forwards evaluation to backwards evaluation. This alters the first options offered, and this option becomes a switch back to forwards evaluation.

Undo Option 8 reverts the last action, whichever it might be (a step forwards or backwards).

Roll back Option 9 shows the user a list of events emitted in the current execution, letting the user revert the execution, one step at a time, until the step after the event was emitted.

Finish evaluation Option 10 ends the process, printing the resulting track to PDF and text files.

The system always enforces causal-consistency, and does not let the user take or undo an invalid step in the execution. Otherwise, it lets them construct and explore the R-track in any fashion.

11.1.1 Use cases

Carl is modelling cryptographic protocols using CSP, but his model does not pass one of the tests. Unsure whether the error stems from the model or the protocol, he traces the execution with *reverCSP* until the error appears, and then performs backward and forward steps to roll back the error and perform the correct sequence. With complete knowledge of the bug and its solution, he patches the model and the tests pass.

[1] A graph visualization suite, available at `https://graphviz.org`.

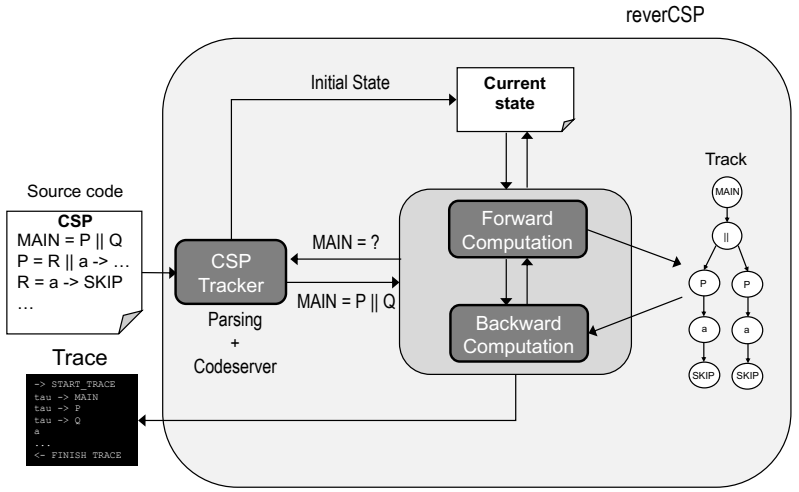


FIGURE 11.2: *reverCSP*'s architecture.

Maya is being introduced to modelling languages like CSP and temporal logic (LTL). She is given the task of describing a model with temporal logic equations. Using *reverCSP*, she generates multiple R-tracks in PDF, which show her the general behaviour of the model. Then, she interactively generates the traces, walking back and forth the track. After interacting with the model, she can posit different LTL formulae to describe its properties and behaviour.

11.2 Architecture

The architecture of *reverCSP* is shown on Fig. 11.2. The software starts its process by reading the CSP source code with the *CSP-tracker* module[2]. It contains functions to parse CSP and execute a single step. It also locates the initial process to be run, and reports that to the pair of forward and backward computation *reverCSP* modules. The default direction (forwards) is started, and it generates the R-track by executing computation steps. To do so, it sends an expression to *CSP-tracker*, which responds with by evaluating a single step. The backwards computation module uses the R-track to perform any backwards step (while remaining causally-consistent). Finally, the generated trace and/or R-track can be written to text and PDF files, respectively. The latter produces a Graphviz specification file (`.dot`), and then it invokes the `dot` layout engine to convert it to PDF.

[2]*CSP-tracker* is an Erlang program and library that can evaluate a CSP specification and generate a track of its execution. An online demo is available at `https://mist.dsic.upv.es/csp_tracker`.

The *src/* folder contains four Erlang modules that implement this behaviour. *reversible_csp.erl* contains the startup logic, argument parsing, and then handles execution to the forwards computation module. After the user quits the program, it saves the trace to a text file and produces a graphical representation in PDF. *reversible_csp_forward.erl* and *reversible_csp_backward.erl* provide the forwards and backwards computation modules, respectively. They determine which steps may be taken at any given time, and delegate the execution to *CSP-tracker*. They store both the current state and R-track. When the direction is switched, the active module calls its counterpart, passing along the current status of the simulation. Functions that are common between the two directions appear in *reversible_csp_lib.erl*. Among others, this modules builds the menu (`ask_questions/3`), dynamically, based on the current state and direction, and processes user input in lieu of the directional computation modules.

Part V

Conclusions and Future Work

Chapter 12

Conclusions

Complexity is inherent to life, just as the universe tends to maximise entropy. With mobile computers—orders of magnitude more powerful than those that brought humans to the Moon—in our pockets, it still takes a few seconds to send a message, or to check incoming email. This is a multi-stakeholder problem, of course. Programmers and developers are not the sole cause of this issue. However, when optimization is not required, it will most likely be skipped, as there are always more important tasks to tackle. Take the example of Grand Theft Auto 5, a videogame from 2013, whose online mode needs more than five minutes to load, up to 15 or 20 minutes depending on the computer. In 2021, a player wondered why that was the case, and tried to get to the bottom of it. With a profiler, they discovered that a single threaded function was trying to parse a 10 MB string, and iterating through it using `strlen` after reading each letter. With a simple change (without having access to the source code), they reduced loading times from 6 to 1.3 minutes. Why did the developer not fix it sooner? The game was wildly popular, with 22 million monthly active players 10 years after its release, thus wasting computing resources on 22 million computers.

Alas, program slicing and other static analysis techniques cannot change developers and companies' priorities, nor should it. However, it can make maintenance easier, so that the little time allotted to it can be efficiently used to better the software we use daily.

This thesis started with the goal of furthering program slicing and related analyses, and we have made strides in a few different areas. Just as this document is split between control and data dependence, we split the contributions made in the same fashion.

The main contributions regarding control dependence in program slicing are:

- Chapter 3 proposes a new kind of control dependence, which is necessary to correctly model the behaviour of *catch* blocks: conditional control dependence [33]. We provide a counterexample to the currently used

representation of *try-catch* statements [4], showing that it can produce incomplete slices from a program with *catch* blocks. Then, we show why no existing $a \rightarrow b$ dependence can properly represent *catch* blocks (i.e., not exclude them or include them in all cases), and we provide the formal definition of conditional control dependence. Afterwards, we integrate this new dependence into the exception-sensitive SDG (ES-SDG), and showcase the process required to build it, passing through intermediate graphs, and showing the representation of exception-related instructions (*throw*, *try*, *catch*...) in each step (ES-CFG, ES-PDG, and ES-SDG). We provide an algorithm to slice the ES-SDG, with additional restrictions to handle conditional control dependences. Then, we provide an empirical evaluation, showing that the construction of the graph is slightly slower with our technique, and that producing slices consumes 20 to 25 % additional time when compared to Allen and Horwitz's algorithm. Our implementation of this technique is available on *JavaSlicer*, which is described in detail on Chapter 9. Finally, we formally proved the completeness of the CE-SDG with regards to exception-related constructs, with an induction-based proof.

- In Chapter 4, we examine the state of the art regarding unconditional jumps (e.g., *break* and *return*), which can be represented with non-executable control flow and, optionally, with pseudo-predicates [32]. First, we analyse the need for non-executable control flow, and its definition. Then, we show which graphs use it, mainly the APDG [8] and the PPDG [62]. The latter introduces a more complex definition of control dependence, in order to produce more precise slices when multiple unconditional jumps are present (such as in a *switch* statement, where most if not all cases end with a *break* or *return*). However, the PPDG introduces an unannounced assumption that must be respected: all control dependence in the graph must originate from its definition, and no alterations (as many techniques do) can be performed. The rest of the chapter is dedicated to show how different techniques break this assumption, and the corrections that can be applied to make them compatible with the PPDG. The representation of calls in the PDG and SDG, the nodes that represent objects and their members in object-oriented slicing, and conditional control dependence (introduced in the previous chapter) are all affected. Finally, we also showcase how our implementation (*JavaSlicer*) tackles this problem, referencing the locations in the code where alterations were made to accommodate the PPDG.
- Chapter 5 takes a small detour away from classic program slicing, to tackle the complexities of concurrency. Instead of working on a

developer-focused programming language, we use CSP, a modelling language, to represent the synchronizations between concurrent events, and the possible interleavings generated by a CSP specification [39]. First, we show how each element in a CSP specification can be uniquely labelled, and each step in a CSP computation traced to produce a reversible track (also known as R-track). Then, we use R-tracks to reverse CSP computations, initially in a deterministic manner (each computation is reversed in exactly the same order), and then causally-consistently (any step can be reversed as long as its consequences have been reversed). To that end, we define a set of functions, and prove that R-tracks allow the user to revert any forward transition with a specific opposite backward transition (and viceversa). We also provide some theoretical details related to our implementation. Practical details such as usage instructions can be found on 11. Finally, we performed an empirical evaluation. It shows the size of the R-tracks and the time taken to produce them. Additionally, we show the time and memory overhead of tracking a CSP computation, compared to simply running it. The results oscillate between one and two orders of magnitude, reaching three orders of magnitude for memory in some benchmarks.

On the other hand, the contributions on data dependence are:

- In Chapter 6, we provide a technique to adequately model concurrent assignments to shared-memory variables across function calls [36]. We start by providing a counterexample to Nanda and Ramesh's tSDG [79], which suffers from a lack of context-sensitivity. We explain the source of this issue, showing their slicing algorithm step by step. Then, we redefine the tSDG from the ground up, doubling up on actual and formal nodes, to represent interferences between calls (i.e., inter-thread flow dependence). We accommodate some elements from Nanda and Ramesh's tSDG, such as timestamps, and provide a formal algorithm to create a tSDG from a SDG. Afterwards, we provide our slicing algorithm, which is context-sensitive, and explore its complexity, w.r.t. the number of nodes, procedures and interferences in the graph. Compared to competing implementations of context-sensitive concurrency, our approach is the most efficient and precise, simultaneously.
- Chapter 7 delves into undecidability, an ever-present issue in static analysis and program slicing. According to Rice's theorem, all non-trivial properties are undecidable, stemming from non-termination and reachability. However, it is still unknown whether many classes of terminating/reachability problems are undecidable. We consider four classes of problems, each dealing with the data dependence analysis of a program

with certain features: with or without routines, and with or without composite data structures. Additionally, we consider three types of data analysis: Reps' data analysis, data-flow analysis, and data dependence analysis. We base our results on Reps' approach [93], which found an equivalence between the Parenthesis-Post's Correspondence Problem or P-PCP (which is undecidable) and a data analysis for programs with routines and composite data structures. First, we adapt Reps' example, and then we remove routines from it, while retaining its parallels to P-PCP. Afterwards, we modify both examples such that not only Reps' data analysis is proved undecidable, but also data-flow and data dependence analyses, by altering the question asked at the end of the program, and thus the analysis required to answer it. By the end of the chapter, we have proved several classes of analysis undecidable for programs with composite data structures.

Furthermore, the only ingredients necessary for these data analyses to be undecidable is for the program to contain (i) a datatype with an infinite domain, (ii) a branching instruction (e.g., *if*), and (iii) a looping construct (e.g., *while* or recursive routines). Composite data structures can be though of as any type with an infinite domain, as any data structure can be encoded in types such as strings or even integers in some languages[1]. Thus, even the simplest programming languages that goes beyond "each variable's value must fit on a given set of computer registers" can produce programs whose data analyses are undecidable.

- Last but not least, Chapter 8 extends our previous field-sensitive techniques to produce a context- and field-sensitive program slicer. Our previous technique, the CE-PDG, uses labels on the graph's arcs to restrict the traversal, storing the structures traversed during slicing (lists, tuples) and their corresponding indices. Since it is incompatible with the SDG, we explore tabular slicing, a technique originally designed for precise data-flow analysis across function bounds. Tabular slicing stores the current and context node instead of just the current node to compute summary arcs on-the-fly. After describing both techniques (the CE-PDG and tabular slicing) in detail, and providing their slicing algorithms, we present our context-sensitive CE-PDG, which is very similar to the CE-PDG, but features a context-sensitive slicing algorithm. For each algorithm, we also provide its asymptotic temporal cost, which goes up throughout the chapter. Finally, we describe the empirical evaluation,

[1]For example, Python does not impose an upper or lower bound on integer values, and are only limited by memory, like strings in most programming languages. In other languages, there normally is a *big integer* data type, such as `Integer` in Haskell or `BigInteger` in Java.

comparing the four possible algorithms: with and without each sensitivity. Each added sensitivity makes the algorithm roughly 10 times slower. Precision increases 10 % when adding field-sensitivity, 2 % for context-sensitivity. When adding both simultaneously, precision increases by 11.8 %, with the downside that the tabular, constrained slicing algorithm is one hundred times slower than the standard PDG algorithm.
We have also implemented the algorithms and techniques described in this chapter in the Erlang slicer *eKnife*, whose usage is described on Chapter 10.

On top of the theoretical contributions backed by formal and empirical evaluations, we have implemented the techniques from Chapters 3, 5, 7 and 8 in different software suites. *JavaSlicer* (Chapter 9), *eKnife* (Chapter 10) and *reverCSP* (Chapter 11) are three tools to slice Java programs, slice Erlang programs, and debug CSP specifications; and they constitute our applied contributions to the field of static analysis and program slicing. All of them are available under free software licenses, which means that our work can be extended by other researchers in academia and the industry, either to use them directly to slice or debug programs or to use them as a library, using the algorithms described in different contexts.

Program slicing is a static analysis tool with many different applications. For some of them, language feature support is important, while others do not require many features. Some require quick slices, others prefer slower slices that are as precise as possible. However, the flagship application, the initial motivation for program slicing is for it to be an aid to debugging, and that application has not been fully realised. IDEs do not include it as a core feature, and plug-ins to add it are few and far between, if they even exist.

With this thesis, we have furthered the theory of program slicing with respect to modern programming constructs, such as exception handling and concurrency, and improved the precision with regard to slicing complex data structures. Program slicing is now more precise overall, it is able to produce complete slices in programs with *catch* statements, and thus it is more useful for all its applications.

Chapter 13

Future Work

There are many aspects in which our work could be expanded. This chapter focuses on avenues to continue each chapter's theme.

Exceptions. The ES-SDG defines most exception-handling structures, but omits *finally* blocks. Multiple techniques to handle it are described by Allen and Horwitz [4], but it has not been formalised. *finally* blocks introduce yet another unconventional control flow pattern: if the *try* block starts, they will always be executed. However, the block can be entered with an exception (thrown in *try*'s body and not caught or in a *catch* block) or without one, and its execution can cause a second exception to activate, which may invalidate our handling of the *active exception* as a single variable. The current benchmarks are quite simple, as *JavaSlicer* is not mature enough to support all features in modern Java programs, and classic benchmarks feature little or no exception-handling structures. We detail key missing features in another section below. We have centred our technique around explicit exceptions (thrown and then caught), but Java also contains implicit exceptions. Theoretically, nothing changes, but our implementation would need to accommodate all these new sources of exceptions (e.g., `1 / 0`, or *NullPointerException* on each call and member access). This should be an optional feature of the slicer, as otherwise each statement would be dependent on all previous statements, given that most lines would contain a source of exceptions. This feature would be less intrusive when paired with a static analysis to determine possible values for variables (denominators that might be 0, objects that might be *null*...), and thus reduce the number of possible exceptions. Finally, this technique could be implemented in *eKnife*, handling Erlang's exceptions. Currently, there is a partial implementation of the theory described, but it has not been enabled and tested.

Unconditional jumps and mixed slicing techniques. This chapter is but a peak at the iceberg of program slicing techniques that can (or cannot)

be combined to handle unrelated concepts, such as concurrency and objects, or exceptions and field-sensitivity. A larger survey is required to categorize the state of the art techniques for each language construct and feature, and analyse possible combinations, unspoken assumptions, incompatibilities, and any other relevant issues. The expected result would be a valuable reference to both check the process in each area of slicing, and also which areas are compatible with each other, and which require further work. This survey would probably be too large, so a subtask would be to ontologically organize novel techniques for program slicing, to locate the current techniques regarding object-oriented programming, concurrency (both with message-passing and shared-memory), field-sensitivity, unconditional jumps, etc.

CSP reversibility. Theoretically, R-tracks provide the necessary flexibility for reversibly debugging CSP specifications, but they do so at great cost, with time overheads ranging from 3 to 100. This is due to the current implementation, which is focused on step-by-step evaluation, and only provides "multiple step" evaluation as a convenience to the user. Thus, a dedicated instrumented evaluation mode that sends the information required to build R-tracks to a concurrent process could improve our technique's usefulness. Other practical improvements to *reverCSP* are detailed on a separate section below.

Concurrency. We describe a context-sensitive approach to interference, but have not discussed the control dependence consequences of synchronizations, such as those induced by a thread waiting on, or joining another. This is a basic requirement for shared-memory concurrency program slicing. Although the timestamps already provide a safeguard against the accidental reordering of events after a synchronization, a more in-depth study of the concept is required. Our technique needs to be implemented into a slicer. Ideally, the target would be *JavaSlicer*, which should be a good fit: Nanda and Ramesh's approach tackled Java. However, we only tackled the data aspect of concurrency, and not the control effects on the programs, mainly those caused by synchronizations, which are fairly common in Java's concurrency model. Thus, a simpler implementation on a language with basic concurrency support, such as the *cobegin-coend* structures used in the chapter might be the correct approach. Once implemented, a thorough empirical evaluation should check our technique, both for completeness and precision, as well as the time overhead required to include context sensitivity. Recall that our approach required additional nodes and arcs, compared to Nanda and Ramesh's, but our slicing algorithm is simpler.

Undecidability. Our analysis concluded with the minimum requirements such that a data analysis might be able to reproduce an instance of P-PCP, and thus be undecidable. These requirements might be further minimised by simplifying the requirements for branching and looping constructs. This would enlarge the set of programs whose data analysis is undecidable. On the other hand, we only considered two mainstream data analysis (data flow and data dependence). This work could be extended to consider other kinds of data analysis, and other static analyses.

Field-sensitive. This technique needs to be validated against larger, real-world Java programs and libraries, to obtain a better view of the time and precision effects of producing context- and field-sensitive slices. Additionally, the expressiveness of access constraints is limited by the requirement to use specific indices. This is an obstacle, as lists and arrays are normally accessed with indices that are not known to static analyses such as program slicing. We have already alluded to this in a footnote, with an asterisk index ($]_*$), but there are more cases to be explored. Finally, we have not described the behaviour of list comprehensions, which require an additional type of constraint to extract each element of a list and use it in an expression to create a new list. Regarding exceptions and field-sensitive slicing, the pattern matching in functional languages produces exceptions when the pattern does not match the expression provided, and that behaviour was not model in this version of the algorithm.

The tools that accompany this thesis are not complete by any means, as they were developed until they could satisfy the requirements for each chapter's empirical evaluation. Thus, each of them can be improved as follows:

JavaSlicer In this slicer, support for *enums* is only partial, and static variables' initialization is not represented in the correct instance of time for data dependences. It is focused on the early Java features, and thus has no specific support for generic type variables or functional elements that appeared in Java 9 and 11, including streams and functional interfaces. Anonymous class support works, but only when no external variables are used (which requires an implicit copy of a reference). Point-to analysis is not implemented, and thus aliasing of object variables is not reflected in the graph. Finally, the interface is quite simple, inputting and outputting `.java` files. A more useful interface would include an integration with an IDE, or even better, with a debugger, skipping lines that are excluded from the slice automatically, or even loading the slice in place of the original file for debugging.

eKnife This slicer suffers from limitations similar to *JavaSlicer*'s. In comparison, however, it is much further along, as most of sequential Erlang has been taken into account. Its biggest limitation when trying to benchmark real-world Erlang programs is concurrency. We have not studied the message-passing concurrency model deeply enough to implement it yet. Its user interface could also be improved, specially the graphs outputted by the tool. Due to the large number of decomposable expressions in the CE-PDG, the number of nodes is too large to manage automatically (through Graphviz), and thus visualising graphs becomes a complicated tasks for any non-trivial Erlang module.

reverCSP This debugger uses a simple console interface, where the user needs an auto-reloading PDF reader to follow the track and its computation (for non-trivial specifications). Thus, the most pressing improvement would be its interface. Additionally, it is the only tool that does not have a matching online demo, given the need to interact we would need to develop a stateful web application or embed it as a WebAssembly program for the user to run in their own browser.

All in all, the promise of program slicing we mentioned in the introduction has not been realised, as there are still many tasks and aspects to be solved. However, we remain optimist that, one day, program slicing will become a useful aid to developers and software engineers throughout the world, simplifying complex programs into manageable slices.

Bibliography

[1] B. Aman et al. “Foundations of Reversible Computation”. In: *Ulidowski I., Lanese I., Schultz U., Ferreira C. (eds) Reversible Computation: Extending Horizons of Computing. RC 2020*. Lecture Notes in Computer Science, vol. 12070. Springer, 2020. DOI: 10.1007/978-3-030-47361-7_1.

[2] Frances E. Allen. “A Basis for Program Optimization”. In: *Information Processing, Proceedings of IFIP Congress 1971, Volume 1 - Foundations and Systems, Ljubljana, Yugoslavia, August 23-28, 1971*. Ed. by Charles V. Freiman, John E. Griffith, and Jack L. Rosenfeld. North-Holland, 1971, pp. 385–390. ISBN: 978-0-7204-2032-6.

[3] Frances E. Allen. “Control Flow Analysis”. In: *SIGPLAN Not.* 5.7 (1970), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479.

[4] Matthew Allen and Susan Horwitz. “Slicing Java Programs That Throw and Catch Exceptions”. In: *SIGPLAN Not.* 38.10 (2003), pp. 44–54. ISSN: 0362-1340. DOI: 10.1145/966049.777394.

[5] Kavya Alluru and Jeganathan. L. *Graph based Data Dependence Identifier for Parallelization of Programs*. 2021. arXiv: 2102.09317 [cs.DC].

[6] T. Altenkirch and J. Grattage. “A functional quantum programming language”. In: *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*. 2005, pp. 249–258. DOI: 10.1109/LICS.2005.1.

[7] Paul Anderson, Thomas Reps, and Tim Teitelbaum. “Design and Implementation of a Fine-Grained Software Inspection Tool”. In: *IEEE Trans. Softw. Eng.* 29.8 (2003), pp. 721–733. ISSN: 0098-5589. DOI: 10.1109/TSE.2003.1223646.

[8] Thomas Ball and Susan Horwitz. “Slicing Programs with Arbitrary Control-flow”. In: *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*. AADEBUG '93. London, UK, UK: Springer-Verlag, 1993, pp. 206–222. ISBN: 3-540-57417-4. DOI: 10.1007/BFb0019410.

[9] Alexis Bernadet and Ivan Lanese. "A Modular Formalization of Reversibility for Concurrent Models and Languages". In: *Proceedings 9th Interaction and Concurrency Experience*. EPTCS. Heraklion, Greece, June 2016. DOI: 10.4204/EPTCS.223.7.

[10] David Binkley et al. "ORBS: Language-Independent Program Slicing". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 109–120. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635893.

[11] Geoffrey Brown and Amr Sabry. "Reversible Communicating Processes". In: *Electronic Proceedings in Theoretical Computer Science* 203 (2016), 45–59. ISSN: 2075-2180. DOI: 10.4204/eptcs.203.4.

[12] L. Cardelli and C. Laneve. "Reversible structures". In: *Proceedings of the 9th International Conference on Computational Methods in Systems Biology*. CMSB '11 321 (2011), pp. 131–140. DOI: 10.1145/2037509.2037529.

[13] Diego Cheda, Josep Silva, and Germán Vidal. "Static Slicing of Rewrite Systems". In: *Proceedings of the 15th International Workshop on Functional and (Constraint) Logic Programming*. WFLP 2006. Elsevier ENTCS 177, 2007, pp. 123–136. DOI: 10.1016/j.entcs.2007.01.010.

[14] Jingde Cheng. "Dependence analysis of parallel and distributed programs and its applications". In: *Proceedings. Advances in Parallel and Distributed Computing*. APDC. 1997, pp. 370–377. DOI: 10.1109/APDC.1997.574057.

[15] Jingde Cheng. "Slicing concurrent programs". In: *Automated and Algorithmic Debugging*. Ed. by Peter A. Fritzson. AADEBUG. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 223–240. ISBN: 978-3-540-48141-6. DOI: 10.1007/BFb0019411.

[16] M.S. Conserva Filhoa et al. "Compositional and local livelock analysis for CSP". In: *Information Processing Letters* 133 (2018), pp. 21–25. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2017.12.011.

[17] Ward Cunningham. "The WyCash portfolio management system". In: *SIGPLAN OOPS Messenger* 4.2 (1992), pp. 29–30. ISSN: 1055-6400. DOI: 10.1145/157710.157715.

[18] V. Danos and J. Krivine. "Reversible communicating systems". In: *Proceedings of the Fifteenth International Conference on Concurrency Theory (CONCUR'04)*. Vol. 3170. LNCS. Springer, 2004, pp. 292–307. DOI: 10.1007/978-3-540-28644-8_19.

[19] Martin Davis. *Computability and Unsolvability. 1982 ed*. Dover Publications, Inc., New York, 1958. ISBN: 978-0-486-61471-7.

[20] P. Degano, R. De Nicola, and U. Montanari. "A partial ordering semantics for CCS". In: *Theoretical Computer Science* 75.3 (1990), pp. 223 –262. ISSN: 0304-3975. DOI: 10.1016/0304-3975(90)90095-Y.

[21] Pierpaolo Degano and Corrado Priami. "Non-Interleaving Semantics for Mobile Processes". In: *Theoretical Computer Science* 216.1–2 (Mar. 1999), 237–270. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(99)80003-6.

[22] E. N. (Mootaz) Elnozahy et al. "A Survey of Rollback- recovery Protocols in Message-passing Systems". In: *ACM Computing Surveys* 34.3 (2002), 375–408. DOI: 10.1145/568522.568525.

[23] Yucheng Fang et al. "Modeling and analysis of the disruptor framework in CSP". In: *Proceedings of the 8th Annual Computing and Communication Workshop and Conference*. CCWC '18. Las Vegas, NV, USA: IEEE Computer Society, 2018, pp. 803–809. ISBN: 978-1-5386-4650-2. DOI: 10.1109/CCWC.2018.8301703.

[24] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Transactions on Programming Languages and Systems* 9.3 (1987), pp. 319–349. DOI: 10.1145/24039.24041.

[25] Amelia C. Fong and Jeffrey D. Ullman. "Finding the depth of a flow graph". In: *Journal of Computer and System Sciences* 15.3 (1977), pp. 300–309. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(77)80032-9.

[26] Carlos Galindo, Sergio Perez, and Josep Silva. "A Program Slicer for Java (Tool Paper)". In: *Software Engineering and Formal Methods*. Ed. by Bernd-Holger Schlingloff and Ming Chai. SEFM. Cham: Springer International Publishing, 2022, pp. 146–151. ISBN: 978-3-031-17108-6. DOI: 10.1007/978-3-031-17108-6_9.

[27] Carlos Galindo, Sergio Pérez, and Josep Silva. "Data Dependencies in Object-Oriented Programs". In: *11th Workshop on Tools for Automatic Program Analysis*. TAPAS. 2020.

[28] Carlos Galindo, Sergio Pérez, and Josep Silva. "Indecidibilidad del análisis de dependencias de datos". In: *Actas de las XXII Jornadas sobre Programación y Lenguajes*. Ed. by L. Panizo. PROLE. Sistedes, 2023. URL: https://hdl.handle.net/11705/PROLE/2023/9797.

[29] Carlos Galindo, Sergio Pérez, and Josep Silva. "Program Slicing Techniques with Support for Unconditional Jumps". In: *Formal Methods and Software Engineering*. Ed. by Adrian Riesco and Min Zhang. ICFEM. Cham: Springer International Publishing, 2022, pp. 123–139. ISBN: 978-3-031-17244-1. DOI: 10.1007/978-3-031-17244-1_8.

[30] Carlos Galindo, Sergio Pérez, and Josep Silva. "Program Slicing with Exception Handling". In: *11th Workshop on Tools for Automatic Program Analysis*. TAPAS. 2020.

[31] Carlos Galindo, Sergio Pérez, and Josep Silva. "Slicing unconditional jumps with unnecessary control dependencies". In: *30th International Symposium on Logic-Based Program Synthesis and Transformation, preliminary proceedings*. Ed. by Maribel Fernández. LOPSTR. 2020, pp. 294–305. URL: https://nms.kcl.ac.uk/maribel.fernandez/LOPSTR2020/Preproceedings.pdf.

[32] Carlos Galindo, Sergio Pérez, and Josep Silva. "Slicing Unconditional Jumps with Unnecessary Control Dependencies". In: *Logic-Based Program Synthesis and Transformation, revised and selected papers*. Ed. by Maribel Fernández. Vol. 12561. Lecture Notes in Computer Science (LNCS). Cham: Springer International Publishing, 2021, pp. 293–308. ISBN: 978-3-030-68446-4. DOI: 10.1007/978-3-030-68446-4_15.

[33] Carlos Galindo, Sergio Pérez, and Josep Silva. "Exception-sensitive program slicing". In: *Journal of Logical and Algebraic Methods in Programming* 130 (2023). ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2022.100832.

[34] Carlos Galindo, Sergio Pérez, and Josep Silva. "Program slicing of Java programs". In: *Journal of Logical and Algebraic Methods in Programming* 130 (2023). ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2022.100826.

[35] Carlos Galindo, Sergio Perez Rubio, and Josep Silva. "Conditional Control Dependence to Represent Catch Statements in the System Dependence Graph". In: *Actas de las XX Jornadas de PROgramación y LEnguajes*. PROLE. SISTEDES, 2021. URL: http://hdl.handle.net/11705/PROLE/2021/005.

[36] Carlos Galindo et al. "Context-sensitive analysis of data interference for concurrent programs". In: *CEUR Workshop Proceedings of the 2023 International Workshop on Petri Nets and Software Engineering*. Vol. 3430. 2023. URL: https://ceur-ws.org/Vol-3430/poster4.pdf.

[37] Carlos Galindo et al. "Field-sensitive program slicing". In: *Journal of Systems and Software* 210 (2024), p. 111939. ISSN: 0164-1212. DOI: 10.1016/j.jss.2023.111939.

[38] Carlos Galindo et al. “ReverCSP: Time-Travelling in CSP Computations”. In: *Reversible Computation*. Ed. by Ivan Lanese and Mariusz Rawski. Cham: Springer International Publishing, 2020, pp. 239–245. ISBN: 978-3-030-52482-1. DOI: 10.1007/978-3-030-52482-1_14.

[39] Carlos Galindo et al. “Reversible CSP Computations”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.6 (2021), pp. 1425–1436. DOI: 10.1109/TPDS.2021.3051747.

[40] Carlos Galindo et al. “Slicing Shared-Memory Concurrent Programs, The Threaded System Dependence Graph Revisited”. In: *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2023, pp. 73–83. DOI: 10.1109/ICSME58846.2023.00019.

[41] Andy Georges, Dries Buytaert, and Lieven Eeckhout. “Statistically rigorous java performance evaluation”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA ’07. Montreal, Quebec, Canada: Association for Computing Machinery, 2007, 57–76. ISBN: 9781595937865. DOI: 10.1145/1297027.1297033.

[42] Elena Giachino, Ivan Lanese, and Claudio Antares Mezzina. “Causal-Consistent Reversible Debugging”. In: *Fundamental Approaches to Software Engineering*. Ed. by S. Gnesi and A. Rensink. FASE. Grenoble, France, 2014, pp. 370–384. DOI: 10.1007/978-3-642-54804-8_26.

[43] Elena Giachino et al. “Causal-consistent rollback in a tuple-based language”. In: *Journal of Logical and Algebraic Methods in Programming* 88 (2017), pp. 99–120. ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2016.09.003.

[44] Herman Heine Goldstine and John Von Neumann. *Planning and coding of problems for an electronic computing instrument, Part II, Volume II*. Institute for Advanced Study Princeton, NJ, 1947.

[45] Anjana Gosain and Ganga Sharma. “Static Analysis: A Survey of Techniques and Tools”. In: *Intelligent Computing and Applications*. Ed. by Durbadal Mandal et al. Vol. 343. New Delhi: Springer India, 2015, pp. 581–591. ISBN: 978-81-322-2268-2. DOI: 10.1007/978-81-322-2268-2_59.

[46] Jurgen Graf. “Speeding Up Context-, Object- and Field-Sensitive SDG Generation”. In: *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. SCAM. 2010, pp. 105–114. DOI: 10.1109/SCAM.2010.9.

[47] Mark Harman, Arun Lakhotia, and David Binkley. "Theory and algorithms for slicing unstructured programs". In: *Information and Software Technology* 48.7 (2006), pp. 549–565. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2005.06.001.

[48] John Hatcliff et al. "A formal study of slicing for multi-threaded programs with JVM concurrency primitives". In: *Static Analysis: 6th International Symposium, SAS'99 Venice, Italy, September 22–24, 1999 Proceedings*. Springer. 1999, pp. 1–18. DOI: 10.1007/3-540-48294-6_1.

[49] Christian Hermanns and Herbert Kuchen. "Hybrid Debugging of Java Programs". In: *Software and Data Technologies*. Ed. by María José Escalona, José Cordeiro, and Boris Shishkov. Vol. 303. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2013, pp. 91–107. ISBN: 978-3-642-36176-0. DOI: 10.1007/978-3-642-36177-7_6.

[50] David Hilbert and Wilhelm Ackermann. *Grundzüge der theoretischen Logik*. Grundlehren der mathematischen Wissenschaften. Springer-Verlag Berlin Heidelberg, 1938. ISBN: 978-3-662-41928-1.

[51] C. A. R. Hoare. *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985. ISBN: 0-13-153271-5.

[52] Susan Horwitz, Thomas Reps, and David Binkley. "Interprocedural Slicing Using Dependence Graphs". In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. Atlanta, Georgia, USA: ACM, 1988, pp. 35–46. ISBN: 0-89791-269-1. DOI: 10.1145/53990.53994. URL: http://doi.acm.org/10.1145/53990.53994.

[53] Susan Horwitz, Thomas Reps, and David Binkley. "Interprocedural Slicing Using Dependence Graphs". In: *ACM Transactions Programming Languages and Systems* 12.1 (1990), pp. 26–60.

[54] Susan Horwitz, Thomas Reps, and David Binkley. "Interprocedural Slicing Using Dependence Graphs". In: *ACM Transactions Programming Languages and Systems* 12.1 (1990), pp. 26–60. ISSN: 0164-0925.

[55] S. Jiang et al. "Improving the Preciseness of Dependence Analysis Using Exception Analysis". In: *2006 15th International Conference on Computing*. IEEE, 2006, pp. 277–282. DOI: 10.1109/CIC.2006.40.

[56] H. Jie, J. Shu-juan, and 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)*. 2011, pp. 883–886. DOI: 10.1109/MEC.2011.6025605.

[57] David A. Kinloch and Malcolm Munro. "Understanding C programs using the Combined C Graph representation". In: *Proceedings 1994 International Conference on Software Maintenance*. 1994, pp. 172–180.

[58] Bogdan Korel and Janusz Laski. "Dynamic slicing of computer programs". In: *Journal of Systems and Software* 13.3 (1990), pp. 187–195. ISSN: 0164-1212. DOI: https://doi.org/10.1016/0164-1212(90)90094-3.

[59] Jens Krinke. "Context-Sensitive Slicing of Concurrent Programs". In: *SIGSOFT Softw. Eng. Notes* 28.5 (2003), pp. 178–187. ISSN: 0163-5948. DOI: 10.1145/949952.940096. URL: https://doi.org/10.1145/949952.940096.

[60] Jens Krinke. "Static Slicing of Threaded Programs". In: *SIGPLAN Not.* 33.7 (July 1998), pp. 35–42. ISSN: 0362-1340. DOI: 10.1145/277633.277638. URL: http://doi.acm.org/10.1145/277633.277638.

[61] Jens Krinke and Gregor Snelting. "Validation of measurement software as an application of slicing and constraint solving". In: *Information and Software Technology* 40.11 (1998), pp. 661 –675. ISSN: 0950-5849. DOI: https://doi.org/10.1016/S0950-5849(98)00090-1. URL: http://www.sciencedirect.com/science/article/pii/S0950584998000901.

[62] Sumit Kumar and 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)*. Vol. 2306. Lecture Notes in Computer Science (LNCS). Springer, 2002, pp. 96–112. ISBN: 3-540-43353-8.

[63] P.B. Ladkin and B.B. Simons. "Static Deadlock Analysis for CSP-Type Communications". In: *In: Fussell D.S., Malek M. (eds) Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems. The Springer International Series in Engineering and Computer Science, vol 297. Springer, Boston, MA*. 1995.

[64] R. Landauer. "Irreversibility and heat generation in the computing process". In: *IBM Journal of Research and Development* 5 (1961), pp. 183–191.

[65] William Landi. "Undecidability of Static Analysis". In: *ACM Lett. Program. Lang. Syst.* 1.4 (1992), 323–337. ISSN: 1057-4514. DOI: 10.1145/161494.161501. URL: https://doi.org/10.1145/161494.161501.

[66] I. Lanese et al. "Controlling Reversibility in Higher-Order Pi". In: *In: Katoen JP., König B. (eds) CONCUR 2011 – Concurrency Theory. CONCUR 2011. Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-23217-6_20*. Vol. 6901. Springer, 2011.

[67] Ivan Lanese, Claudio Antares Mezzina, and Francesco Tiezzi. "Causal-Consistent Reversibility". In: *Bulletin of the EATCS* 114 (2014), p. 17.

[68] Ivan Lanese, Adrian Palacios, and Germán Vidal. "Causal-Consistent Replay Debugging for Message Passing Programs". In: *In: 39th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2019). Copenhagen, Denmark.* 2019, pp. 167–184.

[69] Ivan Lanese et al. *A Theory of Reversibility for Erlang*. 2018. arXiv: 1806.07100 [cs.PL].

[70] Ivan Lanese et al. "CauDEr: A Causal-Consistent Reversible Debugger for Erlang". In: *Functional and Logic Programming* 10818 (2018), pp. 247–263.

[71] Loren Larsen and Mary J. Harrold. "Slicing Object-Oriented Software". In: *Proceedings of the 18th international conference on Software engineering*. ICSE '96. Berlin, Germany: IEEE Computer Society, 1996, pp. 495–505. ISBN: 0-8186-7246-3. URL: http://dl.acm.org/citation.cfm?id=227726.227837.

[72] Donglin Liang and Mary J. Harrold. "Slicing Objects Using System Dependence Graphs". In: *Proceedings of the International Conference on Software Maintenance*. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 358–367. ISBN: 0-8186-8779-7. URL: http://dl.acm.org/citation.cfm?id=850947.853342.

[73] Joseph Libby and Kenneth Kent. *A survey of data dependence analysis techniques for automated parallelization*. Tech. rep. Faculty of Computer Science, University of New Brunswick, 2007.

[74] Shay Litvak et al. "Field-Sensitive Program Dependence Analysis". In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE '10. Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, 287–296. ISBN: 9781605587912. DOI: 10.1145/1882291.1882334. URL: https://doi.org/10.1145/1882291.1882334.

[75] Marisa Llorens et al. "Dynamic Slicing of Concurrent Specification Languages". In: *Parallel Computing* 53 (2016), pp. 1–22. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/j.parco.2016.01.006. URL: http://www.sciencedirect.com/science/article/pii/S0167819116000363.

[76] Marisa Llorens et al. "Tracking CSP computations". In: *Journal of Logical and Algebraic Methods in Programming* 102 (2019), pp. 138–175.

[77] Ahmed Maghawry et al. "A Survey on Program Analysis and Transformation". In: *Computational Statistics and Mathematical Modeling Methods in Intelligent Systems*. Ed. by Radek Silhavy, Petr Silhavy, and Zdenka Prokopova. Cham: Springer International Publishing, 2019, pp. 110–124. ISBN: 978-3-030-31362-3.

[78] Steven Muchnick et al. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[79] Mangala Gowri Nanda and S. Ramesh. "Interprocedural Slicing of Multithreaded Programs with Applications to Java". In: *ACM Trans. Program. Lang. Syst.* 28.6 (2006), 1088–1144. ISSN: 0164-0925. DOI: 10.1145/1186632.1186636. URL: https://doi.org/10.1145/1186632.1186636.

[80] Mangala Gowri Nanda and S. Ramesh. "Slicing Concurrent Programs". In: *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '00. Portland, Oregon, USA: Association for Computing Machinery, 2000, 180–190. ISBN: 1581132662. DOI: 10.1145/347324.349121. URL: https://doi.org/10.1145/347324.349121.

[81] Karl J. Ottenstein and Linda M. Ottenstein. "The Program Dependence Graph in a Software Development Environment". In: *SIGSOFT Software Engineering Notes* 9.3 (1984), pp. 177–184. ISSN: 0163-5948. DOI: 10.1145/390010.808263. URL: http://doi.acm.org/10.1145/390010.808263.

[82] Monali Patil and Vandana Jagtap. "Survey of Different Data Dependence Analysis Techniques". In: *International Journal of Advanced Engineering, Management and Science* 2.7 (2016), p. 239552.

[83] R. Perera, D. Garg, and J. Cheney. "Causally consistent dynamic slicing". In: *Proceedings of the 2016 International Conference on Concurrency (CONCUR'16)*. Ed. by J. Desharnais and R. Jagadeesan. Vol. 59. LIPIcs. chloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 18:1—18:15.

[84] I. Phillips, I. Ulidowski, and S. Yuen. "A Reversible Process Calculus and the Modelling of the ERK Signalling Pathway". In: *Proceedings of the 4th International Conference on Reversible Computation (RC'12)*. LNCS 7581. Springer, 2012, pp. 218–232. ISBN: 978-1-4503-2966-8. DOI: 10.1007/978-3-642-36315-318..

[85] Keshav Pingali and Gianfranco Bilardi. "Optimal Control Dependence Computation and the Roman Chariots Problem". In: *ACM Trans. Program. Lang. Syst.* 19.3 (1997), 462–491. ISSN: 0164-0925. DOI: 10.1145/256167.256217. URL: https://doi.org/10.1145/256167.256217.

[86] Emil L. Post. "A variant of a recursively unsolvable problem". In: *Bulletin of the American Mathematical Society* 52 (1946), pp. 264–268. ISSN: 1088-9485. DOI: 10.1090/S0002-9904-1946-08555-9. URL: https://doi.org/10.1090/S0002-9904-1946-08555-9.

[87] Prakash Prabhu, Naoto Maeda, and Gogul Balakrishnan. "Interprocedural Exception Analysis for C++". In: *Proceedings of the 25th European Conference on Object-oriented Programming*. ECOOP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 583–608. ISBN: 978-3-642-22654-0. URL: http://dl.acm.org/citation.cfm?id=2032497.2032536.

[88] Sergio Pérez Rubio. "Analysis Techniques for Software Maintenance". PhD thesis. Universidad Politécnica de Valencia, 2023. DOI: 10.4995/Thesis/10251/193146.

[89] Xiaofang Qi and Zhenliang Jiang. "Precise Slicing of Interprocedural Concurrent Programs". In: *Front. Comput. Sci.* 11.6 (2017), 971–986. ISSN: 2095-2228. DOI: 10.1007/s11704-017-6189-3. URL: https://doi.org/10.1007/s11704-017-6189-3.

[90] G. Ramalingam. "The Undecidability of Aliasing". In: *ACM Trans. Program. Lang. Syst.* 16.5 (1994), 1467–1471. ISSN: 0164-0925. DOI: 10.1145/186025.186041. URL: https://doi.org/10.1145/186025.186041.

[91] G. Ramalingam, John Field, and Frank Tip. "Aggregate Structure Identification and Its Application to Program Analysis". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, 1999, 119–132. ISBN: 1581130953. DOI: 10.1145/292540.292553. URL: https://doi.org/10.1145/292540.292553.

[92] Thomas Reps. "Program Analysis via Graph Reachability". In: *Proceedings of the 1997 International Symposium on Logic Programming*. ILPS '97. Port Washington, New York, USA: MIT Press, 1997, 5–19. ISBN: 0262631806.

[93] Thomas Reps. "Undecidability of Context-Sensitive Data-Dependence Analysis". In: *ACM Trans. Program. Lang. Syst.* 22.1 (Jan. 2000), 162–186. ISSN: 0164-0925. DOI: 10.1145/345099.345137. URL: https://doi.org/10.1145/345099.345137.

[94] Thomas Reps, Susan Horwitz, and Mooly Sagiv. "Precise Interprocedural Dataflow Analysis via Graph Reachability". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: Association for

Computing Machinery, 1995, 49–61. ISBN: 0897916921. DOI: 10.1145/199448.199462. URL: https://doi.org/10.1145/199448.199462.

[95] H. G. Rice. "Classes of recursively enumerable sets and their decision problems". In: *Transactions of the American Mathematical Society* 74 (1953), pp. 358–366.

[96] Simone Romano. "Dead Code". In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, pp. 737–742. DOI: 10.1109/ICSME.2018.00092.

[97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997. ISBN: 0136744095.

[98] M. Sharir and A. Pnueli. "Two approaches to interprocedural data flow analysis". In: *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981, pp. 189–233.

[99] Josep Silva. "A Vocabulary of Program Slicing-Based Techniques". In: *ACM Computing Surveys* 44.3 (2012).

[100] S. Sinha and M. J. Harrold. "Analysis of programs with exception-handling constructs". In: *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 1998, pp. 348–357. DOI: 10.1109/ICSM.1998.738526.

[101] I. Souilah, A. Francalanza, and V. Sassone. "A Formal Model of Provenance in Distributed Systems". In: *First Works. on Theory and Practice of Provenance (TAPP'09)* (2009), pp. 1–21.

[102] Johannes Späth, Karim Ali, and Eric Bodden. "Context-, Flow-, and Field-Sensitive Data-Flow Analysis Using Synchronized Pushdown Systems". In: *Proc. ACM Program. Lang.* 3.POPL (2019). DOI: 10.1145/3290361. URL: https://doi.org/10.1145/3290361.

[103] Stack Overflow. *Stack Overflow Developer Survey 2023*. 2023. URL: https://survey.stackoverflow.co/2023/ (visited on 05/02/2023).

[104] Frank Tip. "A Survey of Program Slicing Techniques". In: *Journal of Programming Languages* 3.3 (1995), pp. 121–189.

[105] Neil Walkinshaw, Marc Roper, and Murray Wood. "The Java system dependence graph". In: *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*. 2003, pp. 55–64.

[106] Mark Weiser. "Program Slicing". In: *IEEE Transactions on Software Engineering* 10.4 (1984), pp. 352–357.

[107] J Zhang and CZ Jin. "Control-flow-based static slicing algorithm of threaded programs". In: *Journal of Jilin University* 41.4 (2003), pp. 481–486.

[108] Hongyan Zhao et al. “Modeling and Verifying Storm Using CSP”. In: *Proceedings of the 19th International Symposium on High Assurance Systems Engineering (HASE)*. HASE ’19. Hangzhou, China: IEEE Computer Society, 2019. ISBN: 978-1-5386-8541-9. URL: 10.1109/HASE.2019.00037.

[109] Jianjun Zhao. “Multithreaded dependence graphs for concurrent Java program”. In: *Software Engineering for Parallel and Distributed Systems, International Symposium on*. IEEE Computer Society. 1999, pp. 13–13.

[110] Jianjun Zhao. “Slicing concurrent Java programs”. In: *Proceedings Seventh International Workshop on Program Comprehension*. IWCP. 1999, pp. 126–133. DOI: 10.1109/WPC.1999.777751.

[111] Jianjun Zhao, Jingde Cheng, and K. Ushijima. “Static slicing of concurrent object-oriented programs”. In: *Proceedings of 20th International Computer Software and Applications Conference: COMPSAC ’96*. 1996, pp. 312–320. DOI: 10.1109/CMPSAC.1996.544182.

Part VI

Appendices

Appendix A

Publications in this thesis

This appendix includes the published papers that back up this thesis. In all these papers, the author has participated in full capacity, participating in coordination meetings, writing sections or the whole paper, writing software and running experiments, and participating actively in the formalization of definitions, theorems and their proofs. We have excluded from this list papers included in other theses, such as S. Pérez's [88].

The following sections are sorted chronologically.

A.1 Journal Publications

- Carlos Galindo, Sergio Pérez, and Josep Silva. "Slicing Unconditional Jumps with Unnecessary Control Dependencies". In: *Logic-Based Program Synthesis and Transformation, revised and selected papers*. Ed. by Maribel Fernández. Vol. 12561. Lecture Notes in Computer Science (LNCS). Cham: Springer International Publishing, 2021, pp. 293–308. ISBN: 978-3-030-68446-4. DOI: 10.1007/978-3-030-68446-4_15
- Carlos Galindo et al. "Reversible CSP Computations". In: *IEEE Transactions on Parallel and Distributed Systems* 32.6 (2021), pp. 1425–1436. DOI: 10.1109/TPDS.2021.3051747
- Carlos Galindo, Sergio Pérez, and Josep Silva. "Exception-sensitive program slicing". In: *Journal of Logical and Algebraic Methods in Programming* 130 (2023). ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2022.100832
- Carlos Galindo et al. "Field-sensitive program slicing". In: *Journal of Systems and Software* 210 (2024), p. 111939. ISSN: 0164-1212. DOI: 10.1016/j.jss.2023.111939

A.2 Conference publications

- Carlos Galindo et al. “ReverCSP: Time-Travelling in CSP Computations”. In: *Reversible Computation*. Ed. by Ivan Lanese and Mariusz Rawski. Cham: Springer International Publishing, 2020, pp. 239–245. ISBN: 978-3-030-52482-1. DOI: 10.1007/978-3-030-52482-1_14
- Carlos Galindo, Sergio Pérez, and Josep Silva. “Slicing unconditional jumps with unnecessary control dependencies”. In: *30th International Symposium on Logic-Based Program Synthesis and Transformation, preliminary proceedings*. Ed. by Maribel Fernández. LOPSTR. 2020, pp. 294–305. URL: https://nms.kcl.ac.uk/maribel.fernandez/LOPSTR2020/Preproceedings.pdf
- Carlos Galindo, Sergio Pérez, and Josep Silva. “Program Slicing with Exception Handling”. In: *11th Workshop on Tools for Automatic Program Analysis*. TAPAS. 2020
- Carlos Galindo, Sergio Perez Rubio, and Josep Silva. “Conditional Control Dependence to Represent Catch Statements in the System Dependence Graph”. In: *Actas de las XX Jornadas de PROgramación y LEnguajes*. PROLE. SISTEDES, 2021. URL: http://hdl.handle.net/11705/PROLE/2021/005
- Carlos Galindo, Sergio Pérez, and Josep Silva. “Program Slicing Techniques with Support for Unconditional Jumps”. In: *Formal Methods and Software Engineering*. Ed. by Adrian Riesco and Min Zhang. ICFEM. Cham: Springer International Publishing, 2022, pp. 123–139. ISBN: 978-3-031-17244-1. DOI: 10.1007/978-3-031-17244-1_8
- Carlos Galindo et al. “Context-sensitive analysis of data interference for concurrent programs”. In: *CEUR Workshop Proceedings of the 2023 International Workshop on Petri Nets and Software Engineering*. Vol. 3430. 2023. URL: https://ceur-ws.org/Vol-3430/poster4.pdf
- Carlos Galindo et al. “Slicing Shared-Memory Concurrent Programs, The Threaded System Dependence Graph Revisited”. In: *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2023, pp. 73–83. DOI: 10.1109/ICSME58846.2023.00019
- Carlos Galindo, Sergio Pérez, and Josep Silva. “Indecidibilidad del análisis de dependencias de datos”. In: *Actas de las XXII Jornadas sobre Programación y Lenguajes*. Ed. by L. Panizo. PROLE. Sistedes, 2023. URL: https://hdl.handle.net/11705/PROLE/2023/9797

A.3 Talks and dissemination events

This section includes all talks given on the thesis' topics, regardless of whether the corresponding conference paper was published in proceedings or included in previous sections.

- Presented the work "ReverCSP: Time-Travelling in CSP Computations" on an international conference, Reversible Computations 2020 (RC2020), held in Oslo, Norway.
- Presented the work "Data Dependence for Object-Oriented Programs" on an international workshop, Tools for Automatic Program Analysis (TAPAS 2020), held in Chigaco, USA.
- Presented the work "Program Slicing with Exception Handling" on an international workshop, Tools for Automatic Program Analysis (TAPAS 2020), held in Chigaco, USA.
- Presented the work "Slicing unconditional jumps with unnecessary control dependencies" on the International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2020), held in Bologna, Italy.
- Presented the work "Object Variable Dependencies in Object-Oriented Programs" on a national symposium, Jornadas de Programación y Lenguajes 2021 (PROLE 2021), held in Málaga, Spain.
- Presented the work "Conditional Control Dependence to Represent Catch Statements in the System Dependence Graph" on a national symposium, Jornadas de Programación y Lenguajes 2021 (PROLE 2021), held in Málaga, Spain.
- Presented "Program slicing for modern programs", a short talk as part of the SEFM Summer School 2022, held in Germany, Berlin.
- Presented the work "A Program Slicer for Java (Tool Paper)" on an international conference, Software Engineering and Formal Methods 2022 (SEFM 2022), held in Berlin, Germany.
- Presented the work "Field-Sensitive Program Slicing" on an international conference, Software Engineering and Formal Methods 2022 (SEFM 2022), held in Berlin, Germany.
- Presented the work "Reversible CSP Computations" on a national symposium, Jornadas de Programación y Lenguajes 2022 (PROLE 2022), held in Santiago de Compostela, Spain.
- Presented the work "Indecidibilidad del Análisis de Dependencias de Datos" on a national symposium, Jornadas de Programación y Lenguajes 2023 (PROLE 2023), held in Ciudad Real, Spain.
- Presented the work "Grafos para la fragmentación de programas sensible a los campos que mejoran al usarlos" on a national symposium, Jornadas

de Programación y Lenguajes 2023 (PROLE 2023), held in Ciudad Real, Spain.
- Presented the work "Análisis de Programas Concurrentes con Memoria Compartida Sensible al Contexto" on a national symposium, Jornadas de Concurrencia y Sistemas Distribuidos 2023 (JCSD 2023), held in Tarragona, Spain.
- Presented the work "Context-sensitive analysis of data interference for concurrent programs" on the International Workshop on Petri Nets and Software Engineering (PNSE 2023), held in Lisbon, Portugal.
- Presented the work "Slicing Shared-Memory Concurrent Programs, The Threaded System Dependence Graph Revisited" on the International Conference on Software Maintenance and Evolution 2023 (ICSME 2023), held in Bogotá, Colombia.
- Gave an invited talk to the CREST group about field-sensitive program slicing and the challenges to make it context-sensitive, on December 2023 at University College London (UK).

A.4 List of derived artifacts

Artifact	Type	Location
JavaSlicer	Sources	`https://mist.dsic.upv.es/git/program-slicing/sdg`
	Demo	`https://mist.dsic.upv.es/JavaSlicer/demo`
eKnife	Sources	`https://mist.dsic.upv.es/git/program-slicing/e-knife-erlang`
	Demo	`https://mist.dsic.upv.es/e-knife-constrained`
reverCSP	Sources	`https://github.com/tamarit/reverCSP`

Appendix B

Proof of Theorem 3.1: Completeness of the ES-SDG Slicing Algorithm

We first restate the theorem,

Theorem B.1 (Completeness)**.** *Let P be a program and $G = (N, A)$ its associated ES-SDG. Let node $n_{sc} \in N$ be the node associated with the slicing criterion $\langle s, v \rangle$. $Slice(G, n_{sc})$[1] 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 that the behaviour related to exception-generating and handling constructs produces static backward slices (Definition 2.3). 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 B.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 2.3) with a program with n nested `try-catch` blocks.

[1] This function is defined in Algorithm 4.

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 B.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, and that Algorithm 4 is complete, i.e., it produces static backward slices in all cases. □

B.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 $\mathtt{S}_n$, 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.

B.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 $\mathtt{S}_n$ instruction. Cases B.1, B.2, and B.3 display the behaviour of unconditional exception sources.

```
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 }
```

FIGURE B.1: Three procedures which throw an exception unconditionally, with no exception handling (`f`), complete exception handling (`g`), and partial exception handling (`h`).

Case B.1 (Unconditional exception source, exception not handled)**.** *Consider procedure `f`, declared in lines 1-4 of Figure B.1. It contains a single unconditional exception source, and no exception-catching instructions. Now consider its corresponding ES-SDG, shown in Figure B.2. 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, 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 B.2 (Unconditional exception source, exception completely caught)**.** *Consider procedure `g`, declared in lines 6-15 of Figure B.1. 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 Figure B.3. 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`.*

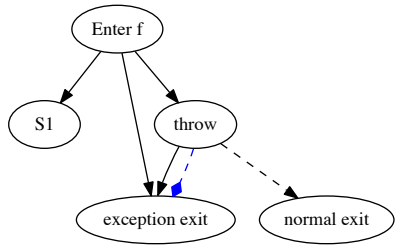


FIGURE B.2: ES-SDG corresponding to procedure `f` in Figure B.1

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.*

Case B.3 (Unconditional exception source, exception partially caught)**.** *Consider procedure* `h`*, declared in lines 17-27 of Figure B.1. 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 Figure B.4. 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 B.2, and therefore the slices are equal. The slices that change are as follows:*

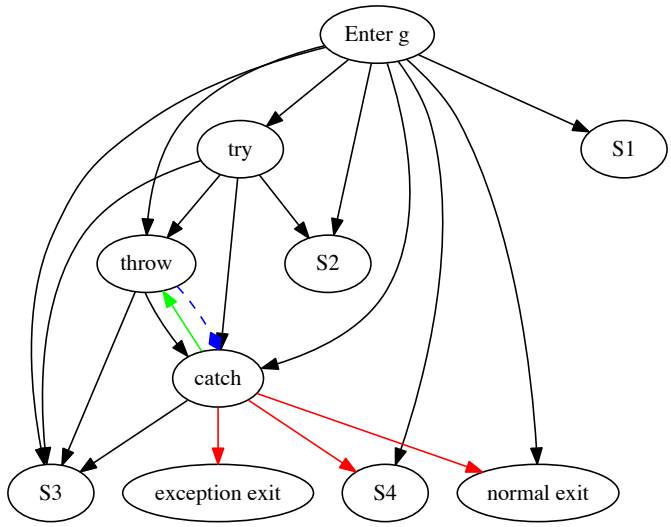


FIGURE B.3: ES-SDG corresponding to procedure g in Figure B.1

- *`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 B.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.*

B.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 $\mathtt{S}_n$-type instructions to analyse the behaviour at all possible points relative to the exception generation and capture. Cases B.4, B.5, and B.6 display the behaviour of conditional exception sources.

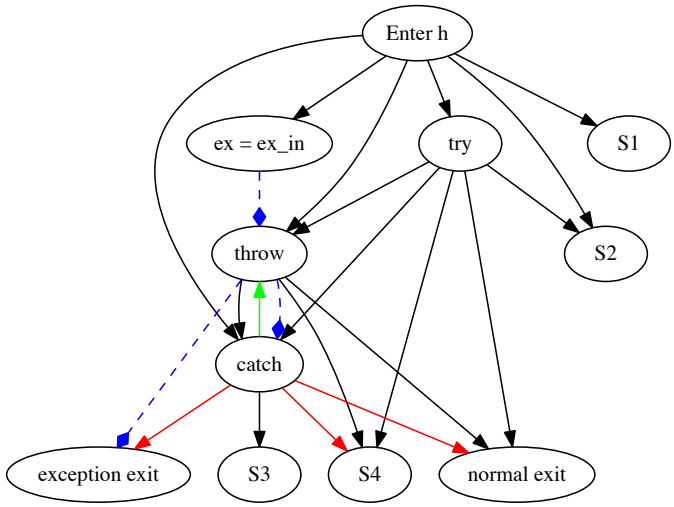


FIGURE B.4: ES-SDG corresponding to procedure `h` in Figure B.1

```
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) {
26     S4;
27   }
28   S5;
29 }
```

FIGURE B.5: Three procedures which throw an exception conditionally, with no exception handling (`f`), complete exception handling (`g`), and partial exception handling (`h`).

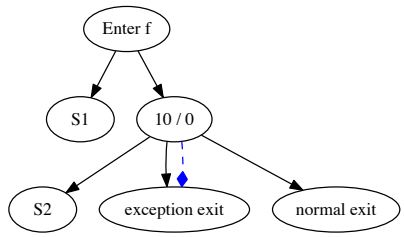


FIGURE B.6: ES-SDG corresponding to procedure `f` in Figure B.5

Case B.4 (Conditional exception source, exception not handled)**.** *Consider procedure `f`, declared in lines 1-5 on Figure B.5. It contains a single conditional exception source, and no exception-capturing instructions. Now consider its corresponding ES-SDG, shown in Figure B.6. Let us consider each node as the slicing criterion and see the resulting slice:*

- 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 B.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 B.5 (Conditional exception source, exception completely caught)**.** *Consider procedure `g`, declared in lines 6-17 on Figure B.5. 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 Figure B.7. When compared to unconditional exceptions, it is again very similar to the corresponding B.2 and Figure B.3: 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.*

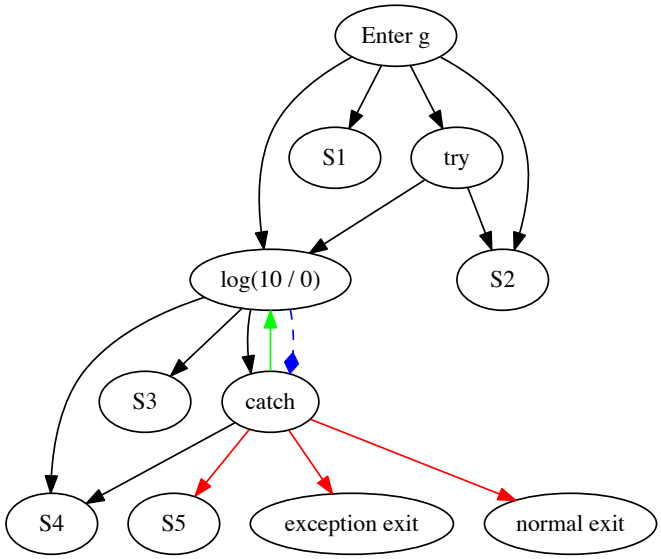


FIGURE B.7: ES-SDG corresponding to procedure g in Figure B.5

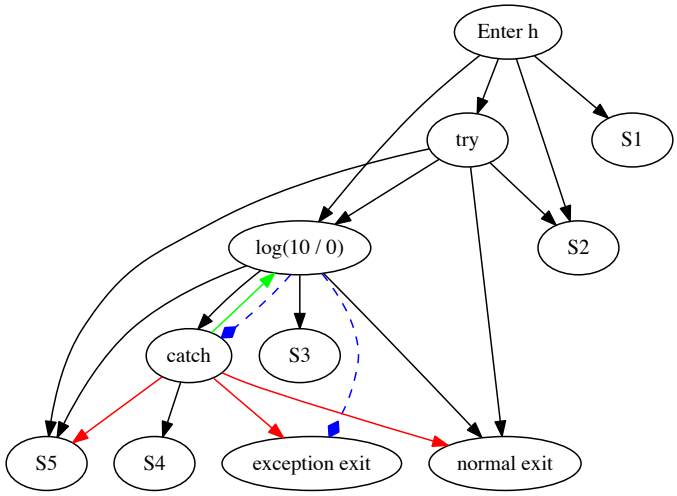


FIGURE B.8: ES-SDG corresponding to procedure `h` in Figure B.5

Case B.6 (Conditional exception source, exception partially caught)**.** *Consider procedure* `h`*, declared in lines 19-29 on Figure B.5. It contains a single conditional exception source, which is partially captured by its surrounding* `try-catch`*. Now consider its corresponding ES-SDG, shown in Figure B.8. As with the previous example (B.5), there are many similarities with its unconditional exception counterpart (B.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 B.3; and the slice of the newly introduced `S3`*:* Enter*,* `try`*, the exception source and itself.*

B.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

```
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 }
```

FIGURE B.9: 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`).

to be able to simulate and select all possible slicing criteria w.r.t. a procedure that may produce exceptions. Cases B.7, B.8, and B.9 display the behaviour of procedure calls that may throw exceptions.

Case B.7 (Procedure that may throw exceptions, exception not handled)**.** *Consider procedure `f`, declared in lines 6-10 on Figure B.9. 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 Figure B.10. The call arc is represented with a dashed arc, and the return arcs are represented with dotted arcs. 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.*

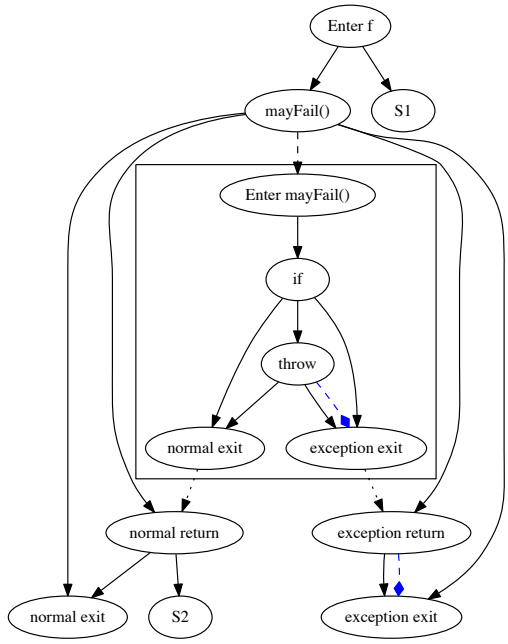


FIGURE B.10: ES-SDG corresponding to procedure `f` in Figure B.9

- Normal exit *or* `S2`*: the slice contains the same nodes as* normal return*'s slice, plus the slicing criterion.*

Case B.8 (Procedure that may throw exceptions, exceptions completely caught)**.** *Consider procedure* `g`*, declared in lines 12-22 on Figure B.9. 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 Figure B.11. The call arc is represented with a dashed arc, and the return arcs are represented with dotted arcs. 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.*

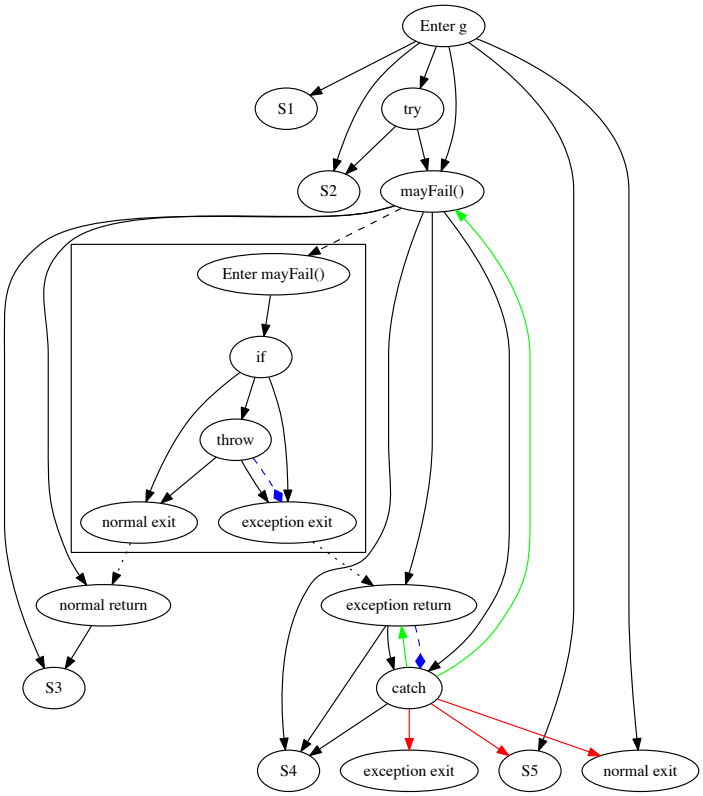


FIGURE B.11: ES-SDG corresponding to procedure g in Figure B.9

- *`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.*

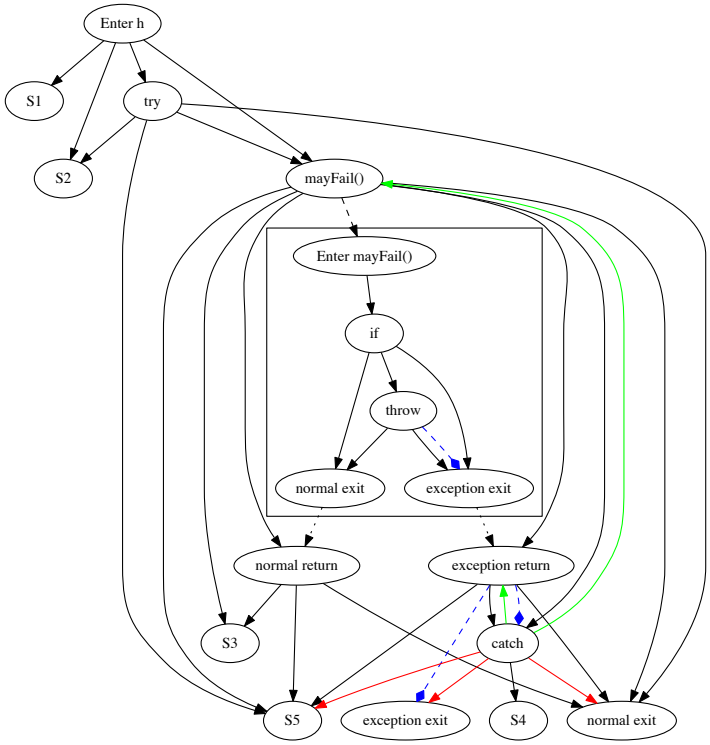


FIGURE B.12: ES-SDG corresponding to procedure `h` in Figure B.9

Case B.9 (Procedure that may throw exceptions, exceptions partially caught)**.** *Consider procedure* `h`*, declared in lines 24-34 on Figure B.9. 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 Figure B.12. The call arc is represented with a dashed arc, and the return arcs are represented with dotted arcs. 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* `mayFail`*,* normal return*,* exception return*,* `catch` *and the slicing criterion. In the case of* exception exit*, notice how the data 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 B.8.*

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

TABLE B.1: All possible combinations for the location where inner and outer exception are caught (either in *inner* `catch` blocks, *outer* `catch` nodes or *none*).

B.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 B.1 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 Figure B.13, where instructions labeled $\texttt{S}_n$ are statements without any effect on control or data dependence; exception sources are displayed as `inner_source` and `outer_source`; and `catch` instructions are labeled `Inner` and `Outer` to be distinguishable. The following sections showcase the ES-SDG produced for each

```
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 }
```

FIGURE B.13: 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.

situation and the slices that it results in.

B.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 Figure B.14. 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

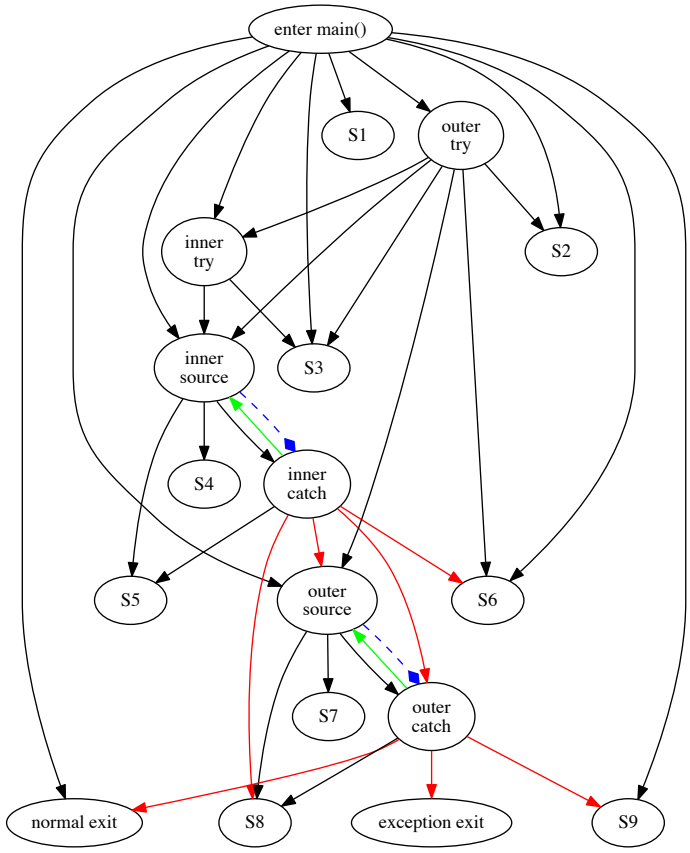


FIGURE B.14: The ES-SDG corresponding to the code in Figure B.13, in the case 1 of Table B.1.

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 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.

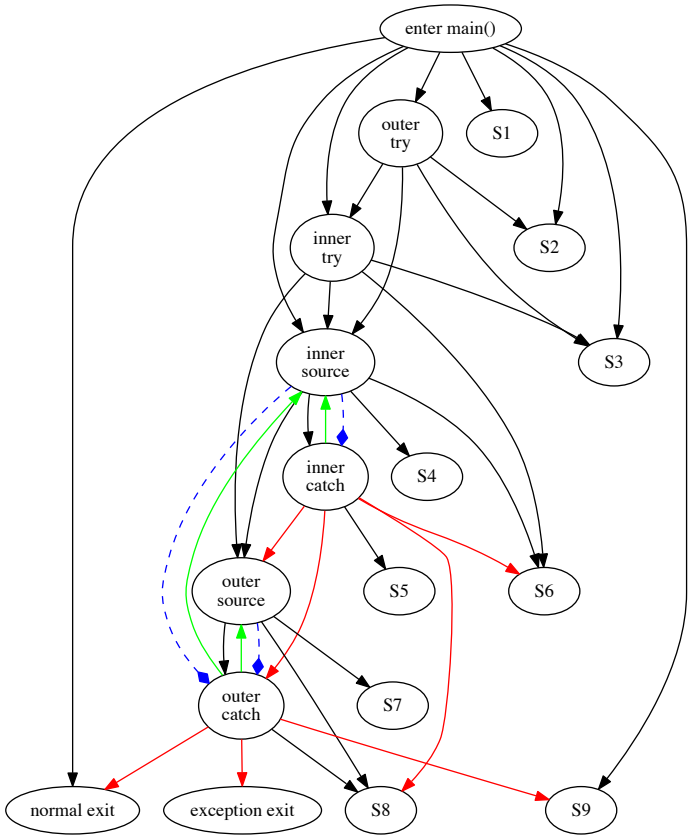


FIGURE B.15: The ES-SDG corresponding to the code in Figure B.13, in the case 2 of Table B.1.

B.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 Figure B.15. 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`.

B.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 Figure B.15. 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.

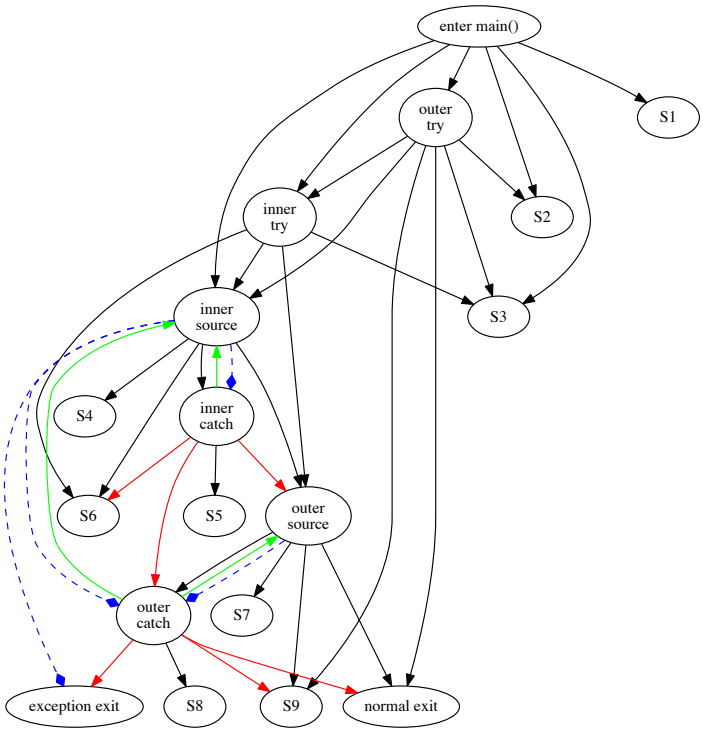


FIGURE B.16: The ES-SDG corresponding to the code in Figure B.13, in the case 3 of Table B.1.

`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.

B.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 Figure B.17. 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`**,** `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

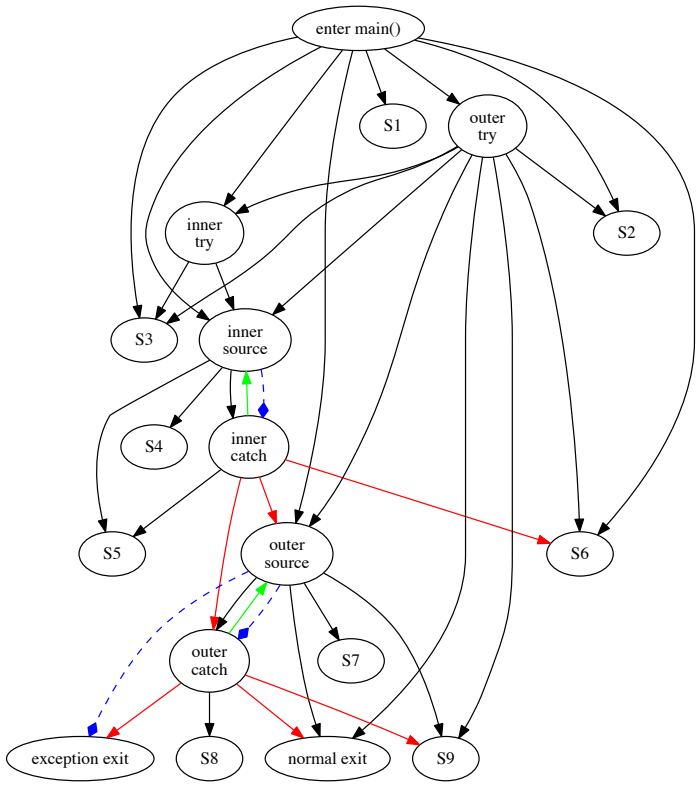


FIGURE B.17: The ES-SDG corresponding to the code in Figure B.13, in the case 4 of Table B.1.

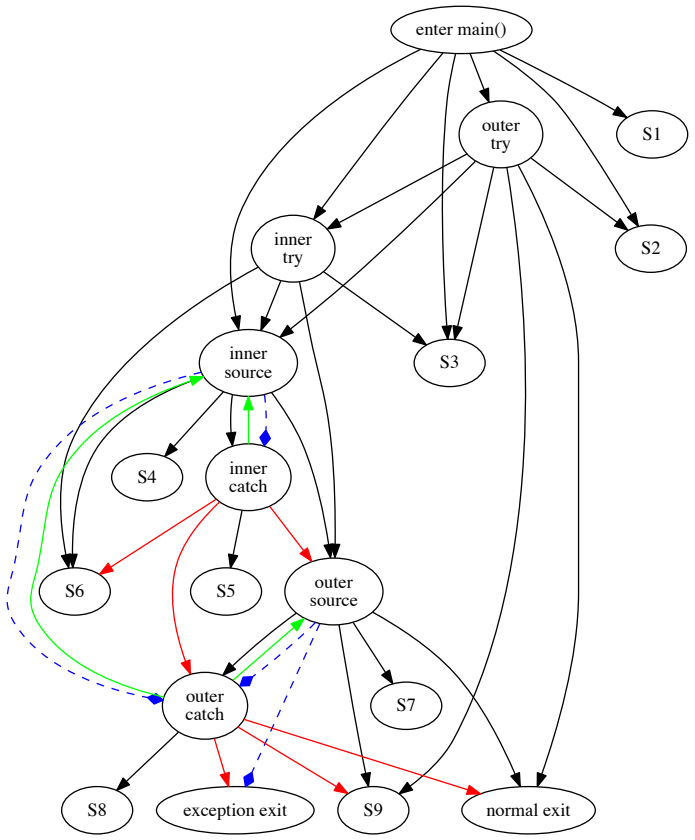


FIGURE B.18: The ES-SDG corresponding to the code in Figure B.13, in the case 5 of Table B.1.

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.

B.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 Figure B.18. 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.

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`.

B.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 Figure B.19. 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.

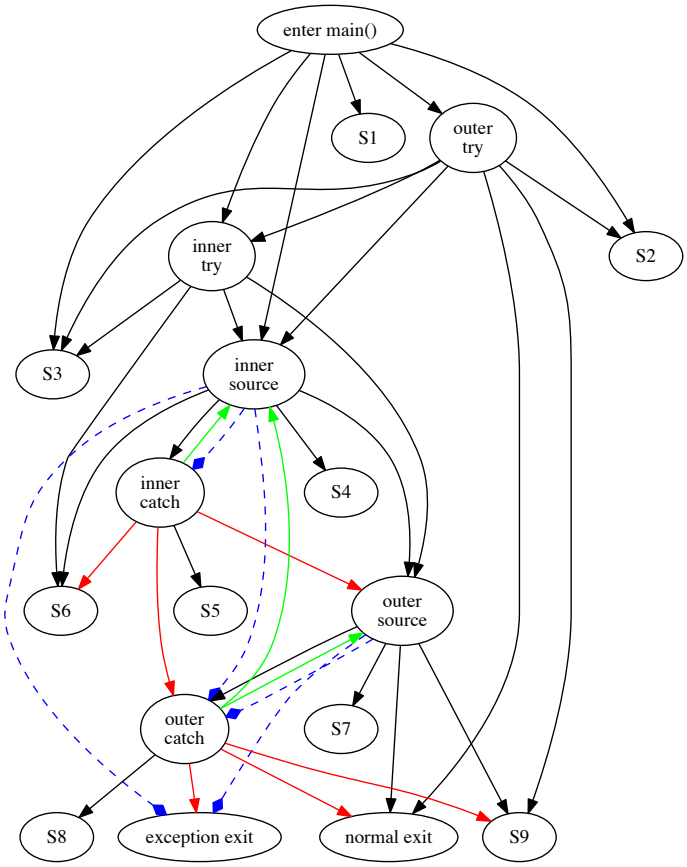


FIGURE B.19: The ES-SDG corresponding to the code in Figure B.13, in the case 6 of Table B.1.

`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 (Figure B.19) and Case 5 (Figure B.18) is the addition of the data dependence between the inner exception source and *exception exit*.