Departamento de Sistemas Informáticos y Computación

Valencian Research Institute for Artificial Intelligence

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

PhD Thesis

Analysis Techniques for Software Maintenance

Author:

Sergio Pérez Rubio

Supervisor:

Josep Silva Galiana

April 2023

Analysis Techniques for Software Maintenance

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Sergio Pérez Rubio

Supervisor

Josep Silva Galiana	Universitat Politècnica de València

Reviewers

Demis Ballis	Università degli Studi di Udine
Miguel Gómez-Zamalloa Gil	Universidad Complutense de Madrid
Konstantinos Sagonas	Uppsala University

Examiners

Demis Ballis	Università degli Studi di Udine
Adrián Riesco Rodríguez	Universidad Complutense de Madrid
Germán Vidal Oriola	Universidad Politécnica de Valencia

“*If you set your goals ridiculously high and it’s a failure, you will fail above everyone else’s success.*”

James Cameron

Abstract

We live in a society where digitalisation is present in our everyday life. We wake up with the alarm of our mobile phone, book our meetings in our digital calendar, save all our media in the cloud storage, and spend time in social networks almost daily. Every one of these tasks is run over a software system that ensures its correct functionality. This massive digitalisation has made software development to shoot up in the last years. During the lifetime of software systems, the maintenance process entails a waste of billions of dollars every year. The cause of this waste is the occurrence of bugs or errors undetected during the software production, which result in a malfunction of the system. For this reason, error detection and localisation techniques, such as testing, verification, or debugging, are a key factor to ensure software quality.

Although many different techniques are used for the debugging, testing, and verification of software systems, this thesis focus on only some of them. In particular, this thesis presents improvements in the program slicing technique (debugging field), a new approach for regression testing (testing field), and a new implementation of the design-by-contract verification model for the Erlang programming language (verification field).

The improvements proposed for the program slicing technique include several enhancements applicable to different scenarios: (i) improvements in the representation and slicing of object-oriented programs, (ii) enhancements in the representation and slicing of (possibly recursive) complex data structures (objects, arrays, lists, tuples, records, etc.), (iii) a new graph model based on a fine-grained representation of programs that augments the expressivity of the graph and provides more accurate slicing results, and (iv) a new technique to compute minimal slices for a program given a set of specific program inputs.

On the other side, the new approach for regression testing is called *point of interest testing*, and it introduces the possibility of automatically comparing the behaviour of any arbitrary point in the code given two versions of the same software system.

Finally, the thesis presents a new implementation of the design-by-contract verification model for the Erlang programming language, where new types of contracts are explained in detail for both the sequential and concurrent parts of Erlang.

All the analyses presented here have been formally defined and their correctness have been proved, ensuring that the results will have the reliability degree needed for real-life systems. Another contribution of this thesis is the implementation of two program slicers for two different programming languages (Java and Erlang), a tool to perform point of interest testing for the Erlang programming language, and a system to run design-by-contract verification in Erlang. It is worth mentioning that all the tools implemented in this thesis are open source and publicly available, so they can be used or extended by any interested researcher.

Resumen

Vivimos en una sociedad donde la digitalización está presente en nuestro día a día. Nos despertamos con la alarma de nuestro teléfono móvil, apuntamos nuestras reuniones en nuestro calendario digital, guardamos nuestros archivos en el almacenamiento en la nube, y entramos a las redes sociales prácticamente a diario. Cada una de estas acciones se ejecuta sobre un sistema software que asegura su correcto funcionamiento. Esta digitalización masiva ha hecho que el desarrollo de software se dispare en los últimos años. Durante el ciclo de vida de un sistema software, la etapa de mantenimiento supone un gasto de billones de dólares anuales. La razón detrás de este gasto es la aparición de bugs o errores que no fueron detectados durante la fase de producción del software, y que se traducen en un mal funcionamiento del sistema. Por este motivo, las técnicas de detección y localización de errores como el testeo, la verificación o la depuración son un factor clave para asegurar la calidad del software.

Aunque son muchas las técnicas que se utilizan para la depuración, testeo y verificación de sistemas software, esta tesis se centra solo en algunas de ellas. En concreto, esta tesis presenta mejoras en la técnica de fragmentación de programas (depuración), una nueva metodología para hacer testeo de regresión (testeo) y una nueva implementación del modelo de verificación de diseño-por-contrato para el lenguaje de programación Erlang (verificación).

Las mejoras propuestas para la técnica de fragmentación de programas incluyen diversas propuestas aplicables a diferentes escenarios: (i) mejoras en la representación y fragmentación de programas orientados a objetos, (ii) mejoras en la representación y fragmentación de estructuras de datos complejas (objetos, vectores, listas, tuplas, registros, etc.), (iii) un nuevo modelo de grafo basado en una representación más detallada de programas, aumentando la expresividad del grafo y generando fragmentos de código más precisos como resultado, y (iv) una nueva técnica para calcular fragmentos mínimos de un programa dado un conjunto específico de posibles valores de entrada.

Por otro lado, la nueva metodología para hacer testeo de regresión se denomina testeo de punto de interés, e introduce la posibilidad de comparar automáticamente el comportamiento de un punto cualquiera del código dadas dos versiones del mismo sistema software.

Por último, la tesis contiene la nueva implementación del modelo de verificación de programas diseño-por-contrato para el lenguaje de programación Erlang, donde se explican en detalle los nuevos tipos de contratos diseñados para las partes secuencial y concurrente de Erlang.

Todos los análisis presentados en esta tesis han sido formalmente definidos y su corrección ha sido probada, asegurando de esta manera que los resultados tendrán el grado de fiabilidad necesario para ser aplicados a sistemas reales.

Otra contribución de esta tesis es la implementación de dos herramientas de fragmentación de programas para dos lenguajes de programación diferentes (Java y Erlang), una herramienta para realizar testeo de punto de interés para el lenguaje de programación Erlang y un sistema para ejecutar verificación de diseño-por-contrato en Erlang. Es de destacar que todas las herramientas implementadas a lo largo del desarrollo de esta tesis son herramientas de código

abierto y públicamente accesibles, de manera que pueden ser usadas o extendidas por cualquier investigador interesado en este area.

Resum

Vivim en una societat on la digitalització està present al nostre dia a dia. Ens alcem amb l'alarma del nostre telèfon mòbil, apuntem les nostres reunions al nostre calendari digital, guardem els nostres arxius al emmagatzematge al núvol, i entrem a las xarxes socials pràcticament a diari. Cadascuna d'aquestes accions s'executa sobre un sistema programari que assegura el seu correcte funcionament. Aquesta digitalització massiva ha fet que el desenvolupament de programari es dispare en els últims anys. Durant el cicle de vida de un sistema programari, l'etapa de manteniment suposa una despesa de bilions de dòlars anuals. La raó darrere d'aquesta despesa és l'aparició de bugs o errors que no van ser detectats durant la fase de producció del programari, i que es traduïxen en un mal funcionament del sistema Per este motiu, les tècniques de detecció i localització d'errors com el testeig, la verificació o la depuració són un factor clau per a assegurar la qualitat del programari.

Encara que són moltes les tècniques utilitzades per a la depuració, testeig i verificació de sistemes programari, esta tesi es centra només en algunes d'elles. En concret, esta tesi presenta millores en la tècnica de fragmentació de programes (depuració), una nova metodologia per a fer testeig de regressió (testeig) i una nova implementació del model de verificació de disseny-per-contracte per al llenguatge de programació Erlang (verificació).

Les millores proposades per a la tècnica de fragmentació de programes inclouen diverses propostes aplicables a diferents escenaris: (i) millores en la representació i fragmentació de programes orientats a objectes, (ii) millores en la representació i fragmentació d'estructures de dades complexes (objectes, vectors, llistes, tuples, registres, etc.), (iii) un nou model de graf basat en una representació més detallada de programes, augmentant l'expressivitat del graf i generant fragments de codi més precisos com a resultat, i (iv) una nova tècnica per a calcular fragments mínims d'un programa donat un conjunt específic de possibles valors d'entrada.

D'altra banda, la nova metodologia per a fer testeig de regressió es denomina testeig de punt d'interés, i introduïx la possibilitat de comparar automàticament el comportament d'un punt qualsevol del codi donades dos versions del mateix sistema programari.

Finalment, la tesi conté la nova implementació del model de verificació de programes disseny-per-contracte per al llenguatge de programació Erlang, on s'expliquen en detall els nous tipus de contractes dissenyats per a les parts seqüencial i concurrent d'Erlang.

Totes les anàlisis presentades en aquesta tesi han sigut formalment definides i la seua correcció ha sigut provada, assegurant d'aquesta manera que els resultats tindran el grau de fiabilitat necessari per a ser aplicats a sistemes reals.

Una altra contribució d'aquesta tesi és la implementació de dos ferramentes de fragmentació de programes per a dos llenguatges de programació diferents (Java i Erlang), una ferramenta per a realitzar testeig the punt d'interés per al llenguatge de programació Erlang i un sistema per a executar verificació de disseny-per-contracte a Erlang. Cal destacar que totes les ferramentes implementades al llarg del desenvolupament d'aquesta tesi són ferramentes de codi

obert i públicament accessibles, de manera que poden ser usades o esteses per qualsevol investigador interessat en el tema.

Agradecimientos

Cuando terminé la carrera de informática tomé una decisión: decidí que quería convertirme en profesor de instituto ya que la docencia siempre me había gustado. Lamentablemente, no lo conseguí, aunque elegir ese camino me llevó a conocer a grandes personas que a día de hoy son muy importantes en mi vida. Tras esa etapa, mientras me replanteaba mis objetivos, una amiga me dijo que un profesor ofertaba un contrato de trabajo en la UPV. Pensé que sería una buena idea seguir cerca del mundo académico y presenté mi currículum. Ese profesor era Germán Vidal, director del grupo MiST de la UPV, la persona que me introdujo en el mundo de la investigación y a quien tengo que agradecer que me colocara en la línea de salida de esta gran etapa de mi vida. Gracias Germán por el voto de confianza que me diste y por toda tu ayuda durante estos años.

Durante el largo periodo que ha sido la realización de mi doctorado, mucha gente se ha mantenido a mi lado de manera incondicional. De hecho, algunas de estas personas lo llevan haciendo durante mucho más tiempo, y son las personas a las que más tengo que agradecer. A mi hermano mayor Vicente, gracias por ser siempre un modelo de referencia y hacer que me esforzara al máximo por seguirte el ritmo. Y por encima de todo, muchas gracias a mis padres, que me lo han dado todo sin pedir nada a cambio, que nunca han cuestionado ninguna de las decisiones que he tomado y que siempre me han levantado de mis caídas y empujado a seguir luchando por lo que quería. Solo puedo daros las gracias, deciros que os quiero muchísimo y, que espero que podáis sentiros tan orgullosos de ser mis padres como me siento yo de ser vuestro hijo.

Muchas son las personas con las que he compartido experiencias y que me han mostrado su apoyo desde que comencé mi doctorado, pero hay una persona que ha sido un faro encendido las 24 horas del día (sobretodo por las noches) y con quien he contraído una deuda que me será imposible pagar, mi director Josep Silva (también conocido como Yoshi). Yoshi, solo tengo palabras de agradecimiento que dedicarte. Gracias por enseñarme el mundo de la investigación, gracias por toda tu paciencia al resolver mis dudas sobre conceptos (muchas veces triviales) que no lograba entender, gracias por guiarme siempre hacia la dirección correcta, gracias por darme a elegir incontables veces en qué quería trabajar, gracias por tu actitud positiva sobre cualquier adversidad que hemos tenido, gracias por criticar siempre mi trabajo de manera constructiva, pero sobretodo gracias por haberme hecho sentir durante todo este tiempo no como un "currela" o un "becario", sino como un igual y un amigo. Ha sido un placer compartir cada día de trabajo durante todos estos años contigo.

Me gustaría también agradecer al resto de los profesores de los grupos ELP y MiST, todos ellos grandes docentes e investigadores que siempre han puesto todo de su parte para ayudarme ante cualquier necesidad. Gracias a todos por vuestros viajes y sus consecuentes dulces. Me gustaría agradecer especialmente a Santiago por ser siempre una fuente de conocimiento investigador y administrativo que tanto me ha ayudado, y a Alicia y Marisa por todos sus consejos y ayuda durante mi incursión al mundo docente, que sin duda fue una de mis mejores experiencias doctorales. Muchas gracias también a mis compañeros doctorandos y doctores del pasado, presente y futuro del ELP y MiST: Nando, Julia, Javier, Lidia, Ángel, Raúl, Víctor, Cristina y Julián, mucho ánimo a todos, espero y deseo que alcancéis vuestras metas en un futuro próximo.

No quisiera olvidarme de aquellas personas fuera de la UPV con las que he trabajado y que me han aceptado en sus “casas” con una hospitalidad indescriptible. Muchas gracias al profesor Marco Comini de la Università degli Studi di Udine, que compartió su tiempo y su conocimiento sobre verificación e interpretación abstracta conmigo durante 3 meses. Muchas gracias también a Lars-Åke, Julio, Clara, Ángel, Nacho y Luís por hacerme sentir como en casa durante mi breve estancia en la Universidad Politécnica de Madrid y por todo lo que me enseñasteis en tan poco tiempo. Gracias también a los revisores y tribunal de la tesis doctoral: Demis, Miguel, Kostis, Adrián y German, cuyos comentarios constructivos han mejorado mucho la calidad de esta tesis.

Cuando pasas tantas horas en un laboratorio durante tantos años con las mismas personas, hay vínculos que se crean sin que tan siquiera te des cuenta. Este ha sido el caso con todos y cada uno de mis compañeros de laboratorio, a los cuales quiero hacer una mención especial. David, la tenacidad y el trabajo duro, que me ha enseñado a buscar los puntos débiles de mis ideas y resolverlos para defenderlas con más fuerza. Adrián, la implicación y la ambición por alcanzar los sueños, que me ha demostrado que comprometerse y seguir tu camino con convicción te llevará a alcanzar el éxito. Tama, la pausa, la crítica y el pensamiento divergente, que me ha mostrado cómo se mantiene la calma en situaciones adversas, cómo analizar las soluciones para identificar los errores y cómo plantear propuestas alternativas, muchas veces explotando estas ideas para resolver otros problemas que podrían estar relacionados. Carlos, el talento y el hambre de conocimiento, que siempre tiene (y si no la tiene, mañana la tendrá) una solución a un problema que tú no consigues resolver por muchas vueltas que le des. No pierdas esa actitud, porque es lo que te hace único. Y por último, Damián, un “intruso” del ELP con alma de becario del MiST. Damián me ha enseñado lo que es la fuerza de voluntad, el cómo todo depende de cuánto estás dispuesto a invertir, porque querer es poder. Damián es el compañero con el que más tiempo he compartido laboratorio, incluso hemos compartido piso los tres meses de estancia en Udine, y no me equivoco si digo que en estos años en los que hemos “sufrido” juntos me llevo de aquí no un compañero, sino un amigo para toda la vida. Espero sinceramente que los “Scaquing Games” que iniciamos todos juntos no terminen aquí, sino que se conviertan en una pretexto para seguir compartiendo momentos en un futuro.

Por último, pero no por ello menos importante, quiero agradecer a todos mis amigos, que han aguantado mis quejas sobre trabajar en la investigación durante nuestras conversaciones de bar todos estos años. Especialmente me gustaría agradecer a Amanda, por acompañarme durante el tramo final de esta etapa de mi vida, que cierra un capítulo que para nosotros no es más que el prólogo de nuestra historia.

Contents

List of Figures

List of Tables

List of Acronyms

ClDG	**Cl**ass **D**ependence **G**raph
CE-EDG	**C**onstrained-**E**dges **E**xpression **D**ependence **G**raph
CE-PDG	**C**onstrained-**E**dges **P**rogram **D**ependence **G**raph
CE-SDG	**C**onstrained-**E**dges **S**ystem **D**ependence **G**raph
CFG	**C**ontrol **F**low **G**raph
DBC	**D**esign-**B**y-**C**ontract
EDG	**E**xpression **D**ependence **G**raph
ITC	**I**nput of a **T**est **C**ase
JSysDG	**J**ava **Sys**tem **D**ependence **G**raph
OO	**O**bject **O**riented
POI	**P**oint **O**f **I**nterest
PDG	**P**rogram **D**ependence **G**raph
QM	**Q**uasi-**M**inimal
SDG	**S**ystem **D**ependence **G**raph
SSLDG	**S**ub-**S**tatement **L**evel **D**ependence **G**raph
TECF	**T**race **E**lement **C**omparison **F**unction
VEF	**V**alue-**E**xtractor **F**unction

Part I

Introduction

Chapter 1

Preamble

1.1 Motivation

At the present time, the world around us is increasingly becoming a technological environment in order to make our lives more comfortable where most devices work thanks to some kind of software system. This is one of the reasons behind the continuous increasing of the production of software, whose market has significantly raised for the last years. Each time a new technological idea comes out of the blue, a journey to develop a software product to cover the expectations begins. When a software project starts, all eyes are usually focused on the specifications and design of the system requirements to transfer the human knowledge to the upcoming system, together with the decision of many other critical factors for software production like estimation of developing times, distinction of critical parts, or selection of the most appropriate developing technologies, among many others.

At the beginning of the software development cycle, the maintenance phase is commonly seen as a far away issue or a non-prior task. Unfortunately, this mindset could not be more wrong. The study published in [102] shows that the Cost of Poor Software Quality (CPSQ) in the USA in 2020 resulted in a cost of \$2.08 trillion. Particularly, \$520 billions were dedicated to software maintenance of legacy systems (i.e., outdated computing software that is still in use). Other studies also state that software maintenance takes approximately between 60-80% of the system and programming resources [73]. This fact is completely captured by the words of the computer science engineer Dan Salomon:

> *"Sometimes it pays to stay in bed on Monday, rather than spending the rest of the week debugging Monday's code."*

These words clearly show how a fragment of code written as part of a system in a couple of hours can result in long testing and debugging sessions in order to properly integrate it to a complex system. The reason of the incredible cost of software maintenance is that programmers, regardless their level of programming skill, most probably will introduce bugs in their programs, usually hidden under complex combination of events that imply different interactions between components that rarely occur in practice. Programmers cannot consider all possible interactions in the programs they write, and those that are unconsidered usually produce a malfunction of some parts of the software system. Furthermore, even if the programmer was able to appropriately account for all program

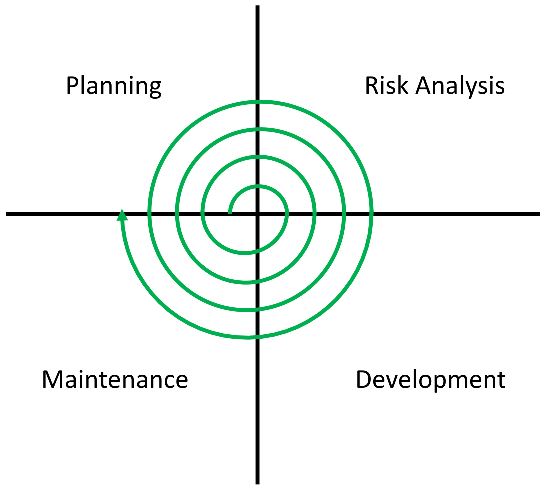


FIGURE 1.1: Spiral representation of the software lifecycle

interactions, it does not prevent a misbehaviour to appear in the future due to the evolution of the system requirements.

Software maintenance is one of the main phases of the spiral software life cycle introduced in [22], represented in Figure 1.1. As the figure shows, the continuous evolution of software implies new analysis and development phases that lead to a new maintenance phase. Although Figure 1.1 shows development and maintenance as two differentiated phases, they are strictly linked in practice and sometimes it is difficult to trace where the line between these two phases is. In fact, the initial testing stages are considered inside software development, while program debugging sessions taken after unveiling any system malfunction are usually considered maintenance. Maintenance includes a set of verification and evaluation processes code must pass, together with the process of code refactoring and, in this phase, code is tested, verified, and then debugged if any error is detected.

The maintenance phase implies different testing methods [141]: *unit testing* to test software components, *functional testing* to test system requirements, *security testing* to test data protection, *stress testing* to test the system performance and support in an extreme level of load, and *regression testing* to test if the integration of new components has introduced any misbehaviour over the already integrated components. Apart from testing, other software verification processes are also used during maintenance to evaluate the reliability of systems like model checking [40, 161] or abstract interpretation [44, 140]. On the other hand, when bugs are detected by testing or verification methods, debugging sessions are run in programming environments with different debugging tools, like trace debuggers to stop the program and analyse the state in user-selected points, letting to execute the program step by step and helping the programmer to trace the source of the error. Some other debugging techniques like abstract debugging [23], or algorithmic debugging [178] can also be used to trace the error, or even static analysis techniques that analyse the dependencies between program statements like program slicing [204] can be handy to remove from the equation parts of the code that surely are not implicated in the error.

Maintenance is one of the most important phases of the software lifecycle that includes all kinds of analyses and verifications over the developed system. In general, maintenance is considered as an activity that extends the lifetime of a system and grants user needs. Software maintenance has been researched for more than 40 years, continuously enriching the topic with new classifications and typologies of maintenance types. The first and the most influential typology of maintenance types was proposed by Swanson in [189], and classifies software maintenance activities into three different types:

- *corrective maintenance*, performed as a consequence of fault location,
- *adaptative maintenance*, done as a result of changes in the system requirements, and
- *perfective maintenance*, carried out in order to enhance maintainability, improve performance, or remove inefficiencies.

Swanson's typology was later used and interpreted by many researchers. For instance, Chapin et al. [31] proposed a refined classification with twelve types of software maintenance activities suggesting that different software organisations can use different types of software maintenance. The standard ISO 14764:2006 [90] divided maintenance in four categories: *corrective*, *adaptive*, *perfective*, and *preventive*. Despite the classification we study, corrective maintenance is commonly the one that attracts the most attention, since it intends to fix discovered problems and brings software to an operational state for end users, acquiring a high priority over other types of work [11].

Along this thesis, we resolve problems of some analysis techniques used in two of the aforementioned maintenance types: *program slicing*, linked to corrective maintenance, and *regression testing* and *design-by-contract verification*, both linked to corrective and perfective maintenance.

1.2 Analysis Techniques

Debugging and testing are considered two of the main pillars of software maintenance. These two processes are linked into an iterative model where new developed code is first tested by a set of testing methods (unit testing, functional testing, performance testing...), and then debugged if any misbehaviour is detected. Afterwards, once debugging locates and corrects any error, the software is re-tested, repeating this cycle until no more errors are found. For this reason, testing and debugging techniques are of great importance in the software evolution process.

Although the techniques used as part of the debugging and testing processes are diverse, in this thesis we particularly focus on three of them: program slicing (used in the debugging phase to ease the location of program errors), regression testing (part of the software integration process of the testing phase), and design-by-contract verification (used during the development phase to verify the correct working of new added software and detect unsupported system executions).

Program Slicing

Program slicing [179, 195] is a technique for program analysis and transformation that answers the following question: "What program statements potentially affect the value of a variable v at a program point p?". The set of statements given as the answer to this question is called the *program slice*, while the point of interest itself ($\langle p, v\rangle$) is known as the *slicing criterion* [147]. In order to obtain the program slice, the program is decomposed by analysing data and control flow between the parts of the program. The technique was initially proposed by Mark Weiser [204] in the context of program debugging of imperative programs. Weiser's initial proposal was to isolate the program statements that may contain a bug to ease the bug location. Since its definition, program slicing has been proved useful in many disciplines such as software maintenance [74], debugging [49], code obfuscation [131], or program specialisation [145], among others.

Let us illustrate the technique by using an example extracted from [195].

Example 1.1. *Figure 1.2a shows a simple program that reads a positive integer number n and then computes the sum and the product of the first n natural numbers. Figure 1.2b shows the slice with respect to the slicing criterion $\langle 11, prod\rangle$ (in underlined blue). As shown in the figure, the slice (in black) only contains the statements that actively contribute to the computation of variable prod, ignoring all the statements implied in the computation of the variable sum (in grey). This notation to represent slices (underlined blue to represent the slicing criterion, black to represent slice, and grey to represent the removed code) will be used along the whole thesis.*

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

(A) An example program

```
read(n);
i = 1;
sum = 0;
prod = 1;
while (i <= n) {
    sum = sum + i;
    prod = prod * i;
    i = i + 1;
}
write(sum);
write(prod);
```

(B) A slice for the program w.r.t. $\langle 11, prod\rangle$

FIGURE 1.2: Example of program slicing

Since program slicing was defined, two different kinds of approaches have been used to compute program slices. The first one was proposed by Weiser in its seminal paper, and it is an approach based on data-flow equations. In this approach, slices are obtained by computing consecutive sets of relevant variables for each node of the *Control Flow Graph* (CFG) [5]. The process initially includes directly relevant statements for each node in the CFG using

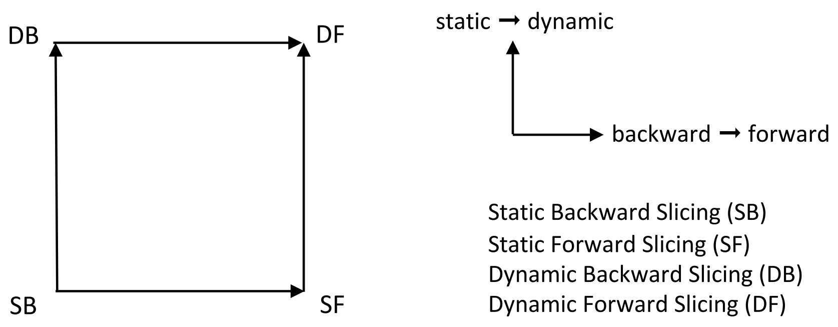

DB
DF
SB
SF
static → dynamic
backward → forward
Static Backward Slicing (SB)
Static Forward Slicing (SF)
Dynamic Backward Slicing (DB)
Dynamic Forward Slicing (DF)

FIGURE 1.3: Dimensions of program slicing

flow dependencies and the indirectly relevant dependencies (variables of control predicates, e.g., variables at `if` and `while` conditions). The process starts from the slicing criterion and repeats until it reaches a fix point (the last iteration does not add any relevant statement to the slice). On the other hand, the most used slicing method is to compute program slices via a graph reachability problem, a method proposed by Ottenstein and Ottenstein in [147]. In this approach, the program is transformed into a set of *Program Dependence Graphs* (PDGs) [56], one graph per program routine (i.e., function, procedure, method...). A PDG is a directed graph where nodes represent program statements and edges represent different kinds of dependences between them. After constructing the PDG, the program slice is computed by traversing all possible edges on the graph starting from the node that represents the slicing criterion.

Weiser's proposal was the seed of a large amount of different approaches to compute program slices. Slicing techniques can be classified according to many different indicators ([179]) but, in general, they can be grouped according to two main dimensions: the existence of *input information*, and the *direction* of the slice.

- **Input information**. According to this dimension, a slicing technique is considered as *static* or *dynamic*. In *static slicing* the slice is computed without having any information about the input of the program, thus, considering every possible program execution. On the other hand, in *dynamic slicing* it is computed with respect to the execution trace given by a particular input.

- **Direction**. According to this dimension, a slicing technique is considered as *backward* or *forward*. In *backward* slicing the slice contains all those statements that influence the values computed at the slicing criterion. On the contrary, in *forward* slicing the slice contains all those statements which may be influenced by the value of the slicing criterion.

Figure 1.3 graphically shows the possibility of combining these two slicing dimensions, classifying the slicing techniques in four different groups. The original program slicing method described by Weiser was *static-backward* so it would be contained in the bottom left corner of the figure. Along this thesis, we describe various slicing approaches related to several programming paradigms and that solve different slicing problems. All the program slicing techniques proposed in this thesis are designed for static-backward slicing.

Regression Testing

Regression testing [162, 185, 210] is a validation technique performed when changes are made to an existing program. According to IEEE, regression testing is "*Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or components still complies with its specified requirements*" [86]. The purpose of regression testing is to check whether the changes performed to some part of the code have not introduced any misbehaviour to the program, and the new code interacts properly with the previously existing code. The common practices of reusing parts of a software project and the use of iterative development strategies increase the need for effective mechanisms for regression testing. Studies indicate that more than 50% of the software maintenance cost is related to testing, being 80% of the testing cost invested in regression testing [38].

Regression testing is performed between two different versions of the code in order to verify that code changes do not interfere with the previous code. Common methods of applying regression testing include the rerun of previously evaluated tests and checking that the behaviour of the program is preserved and no previously solved errors re-emerge. If regression testing exposes any problem, the code must be inspected to check whether the problems are non-detected scenarios of the previous version of the code or they appear due to the changes introduced into the program. Regression testing is a key factor in the software life cycle and is required when:

- The program requirements change and the code is modified accordingly,
- New functionalities are added to the program,
- A bug already existing in a program version is located and fixed, or
- Program changes are introduced in order to improve the performance of some algorithms.

Regression testing requires the existence of a set of tests that run a large amount of program execution paths, testing most of the possible scenarios and detecting misbehaviours in any of them. The generation and maintenance of these test cases is frequently a complex task because many factors need to be considered:

- *Test case selection.* It is hard to assess the impact of changes on existing code making a good selection.
- *Test case design.* When and who should take responsibility of building the regression tests? Some approaches make programmers responsible for writing these tests while developing software.

- *Automatisation of regression testing.* While the application of automating regression testing incurs into problems like the prioritisation of some test cases over others or the automatic generation of new test cases when changes in the program specification are detected, the use of manual testing is time and resource consuming. Therefore, it is difficult to find a balance between both approaches.

- *Test results.* How test results are presented and analysed is a key factor. In many cases test reports are inconsistent and often there is no time to do a deep analysis of the results.

- *Test suite maintenance.* Much of the regression testing is redundant with respect to test coverage. In most cases, there is a lack of good tools for documenting traceability between test cases and requirements.

- *Test frequency.* The lapse of time between regression testing sessions is also difficult to establish. It is commonly good to have a process with a flexible scope for weekly regression tests, although in some cases the size of the regression suite may be a considerably time-consuming task.

Studies show that there is a lack of time and resources for regression testing in practice [50], and a general good practice is to plan for as much test time as development time even when the project is delayed. Some industrial surveys of the application of regression testing have been undertaken [30, 173, 174], concluding that test automation is a key improvement issue [174] and test case selection for continuous regression testing is a hard task. In these surveys, no systematic approach for test case selection was used by the companies but they commonly rely on the developers expertise and judgment instead [173].

Design-by-contract software verification

During the software development process, a set of debugging and testing tools are usually used to find errors before the final software deployment. However, there are still some errors that can escape from the analysis of these tools. These uncontrolled errors can appear even when the program is being used by the final user. In most cases, these errors have their source in a knowledge that programmers had when they were writing the code, e.g., that the result of a function must be greater than the first parameter. Unfortunately, programming languages rarely provide a method to input this information.

Design-by-contract (DBC) is a software correctness methodology that makes the use of this information possible. DBC follows the principle that interfaces which communicate different modules in a software system should follow a set of precise specifications, as it happens in contracts between humans or companies. The contracts must cover mutual obligations (preconditions), benefits (postconditions), and consistency constraints (invariants). Together, these properties are known as *assertions*, and they are directly supported in some design and programming languages. One of the principles of DBC is that any software element that has a fundamental constraint should state it explicitly, as part of a mechanism present in the language. Additionally, assertions are a key component of the software documentation, so their specification entails the presence of this information into the documentation automatically generated from the code.

subsubsectionProgramming languages used

The techniques proposed in this thesis are heterogeneous and sometimes they are only applicable to a specific programming paradigm. For this reason, our proposals use different programming languages according to the problem they are trying to solve. The two programming paradigms we deal with are the object-oriented paradigm and the functional paradigm, represented by the *Java* and *Erlang* programming languages, respectively. We briefly describe the main features of both languages hereunder:

- **Java** [12] is a high-level, general-purpose, concurrent, class-based, object-oriented programming language. Java is intended to let the WORA (Write Once, Run Anywhere) philosophy, meaning that compiled Java code can run in any platform that gives support to Java without recompiling it. Java programs are compiled to an intermediate program representation called *Java bytecode*, that can be run on any Java virtual machine regardless of the underlying computer architecture. The Java language is strongly typed, thus, its specification clearly differentiates between compile-time errors and runtime errors. At runtime, Java provides dynamic capabilities (such as reflection and runtime code modification) that are typically not available in traditional compiled languages. As of 2021 [148], Java was one of the most popular programming languages selected by developers (34.51% of developers write Java code in their programs), particularly for client-server web applications.

- **Erlang** [201] is a concurrent, functional programming language based on the actor model [79]. This language has many distinguishing features like concurrency via asynchronous message passing or hot code swapping, which make it especially appropriate to build massively scalable soft real-time systems with requirements on high availability. The sequential subset of Erlang is a functional language, with strict evaluation, single assignment, and dynamic typing. The term Erlang is used interchangeably with Erlang/OTP, or Open Telecom Platform (OTP), which consists of the Erlang runtime system, several ready-to-use components mainly written in Erlang, and a set of design principles for Erlang programs.

1.3 Contributions and Main Goals

The main goal of this thesis is to improve specific techniques that are used in the maintenance process of the software development cycle. The techniques described in this thesis are used during the development and bug detection/location phases. These techniques are diverse, some working at a static level when running a refactoring over the code or debugging a program after the detection of a bug, and others at a dynamic level to detect the existence of program errors or incorrect program states. The static analysis techniques are mainly focused on the area of program slicing, dealing with some slicing limitations, while the dynamic analysis techniques are focused on software testing and verification, concretely in the areas of regression testing and design-by-contract runtime verification.

Program slicing

Most program slicing techniques have the PDG in their basis, but the original definition of the PDG is not able to handle all the features that most modern programming languages offer. Therefore, numerous extensions and enhancements of the PDG have been proposed to represent different programming features such as arbitrary control-flow [65, 109], exception handling [6, 64, 93], interprocedural behaviour [17, 41], or concurrency [36, 104] among others. In the program slicing area, new proposals are frequently presented as an evolution of a previously existing program representation that lacks the expressiveness needed to manage a particular program feature. Very often, the program representation used as a basis is the PDG; and it is augmented (and commonly renamed) to provide a new representation that deals with the new program feature. Consequently, when further research over the same topic is done, the base representation used changes from the PDG to the last existing representation. Then, when researchers detect unexpected scenarios that cannot be solved by the representation, a new one is proposed, causing the last representation to evolve. In this thesis we deal with different program slicing scenarios, alternatively using the PDG or a PDG-derived program representation as the base of our proposals. The choice of the representation used in each case is conscientiously done by selecting the most suitable representation for the approach we propose in each case. The program slicing scenarios we deal with along this thesis are the following:

- **Program slicing of object-oriented programs.** We augment the expressivity of a previous graph representation used to deal with object-oriented programs, the JSysDG ([202]). In this research, we replace the current definition of flow dependence with three more accurate definitions: the standard definition of flow dependence for primitive variables and another pair of new definitions for object variables that we call *object-flow dependence* and *object-reference dependence*. The inclusion of the dependences derived from these definitions allows any object variable of the program to be selected as the slicing criterion, including in the slice the necessary statements to compute its whole value (the value of its memory reference and the value of all its data members) and to obtain complete slices.

- **Fine-grained program slicing.** The main objective of this research is to formally define a new graph representation with a greater granularity level than the PDG, the *expression dependence graph* (EDG). The granularity solves in a natural way some representation problems of the PDG, like the split of the PDG representation introduced for *for* loops, or the representation of multiple variable definitions in the same statement. The main contribution is the formal definition of the EDG, and the proof that it is a generalisation of the PDG. This definition includes several new ideas that gave rise to different contributions:

  - The identification and formal definition of a new intra-statement dependence called *value-dependence*.
  - The formal definition of *declaration dependence*, which accounts for the declaration-definition relationship, besides the definition-use relationship (flow dependence).
  - The introduction of a new *expression-result* structure to represent expressions. It allows us to distinguish between an expression and its result (e.g., as the slicing criterion).

- **Program slicing of composite data structures.** We design a new proposal to deal with the representation and slicing of composite data structures. The method is applicable to any composite data structure. In this proposal (i) we expand the PDG representation for those statements that represent data structures, (ii) we label the PDG edges with structural information about the data structures, and, finally, (iii) the labels are used at slicing time to limit the slicing traversal, generating more accurate program slices.

- **Testing of program slicers.** Since the computation of a minimal slice is undecidable, we introduce a methodology to compute *quasi-minimal* slices, which are minimal slices w.r.t. a finite set of inputs. These programs and their *quasi-minimal* slices are used afterwards to measure the quality of a program slicer by executing it against them and comparing the produced output and the previously computed slice. The process to obtain *quasi-minimal* slices makes use of the observation-based slicing technique (further explained later in Section 2.4). The main contributions of this part are:

  - A method to compute *quasi-minimal* slices.
  - An adaptation of *observation-based slicing* (ORBS) to work with abstract syntax trees (ASTs). This maximises precision, allowing us to slice at the level of literals.
  - A generalisation of ORBS. The algorithm of ORBS is not well defined for all cases. This problem was identified and solved in our approach.
  - A suite of benchmarks with challenging program slicing problems together with their *quasi-minimal* slices. The suite includes a tool that can be used to evaluate a program slicer against the suite.

Finally, we have implemented different slicing tools that incorporate the four described proposals, which are also contributions for the slicing community. These implementations are done for different programming languages and more information about the tools and how to use them can be found in Chapter 10.

Testing and verification

In the area of testing and verification, our contributions are mainly focused on two main topics:

- **Point-of-interest testing.** We introduce the ability to specify *points of interest* (POI) in the context of testing to compare the behaviour of different program versions. POI testing generates a set of possible calls to a given function and it compares that the POIs of the different code versions generate the same sequences of values for each execution.

  The main contributions of this part are:

  - A methodology to compare the sequence of values generated for two arbitrary points in two different program versions.
  - A technique to automatically generate a set of input values for a particular program function that maximises branch coverage.
  - A set of program transformation rules that modify an Erlang program to extract the value of an arbitrary point of the program as a side-effect of the execution.
  - A tool called `SecEr` that implements the POI testing approach for Erlang.

- **Design-by-contract verification in Erlang.** We implement the first library that implements design-by-contract approaches in Erlang, the EDBC (Erlang Design-By-Contract) library. The library includes two different kinds of contracts: a set of contracts directed to every Erlang program (pre-condition contracts, post-condition contracts...), and another set of contracts specifically designed for concurrent programming (contracts associated to some Erlang behaviours, like the `gen_server` behaviour).

1.4 Structure of this thesis

This PhD thesis is divided into four main parts: Introduction, Program Slicing, Testing & Verification, Developed Tools, and Conclusions and Future Research.

1. The Introduction part only contains Chapter 1. It explains the main motivation of this thesis and provides an introductory description of the analysis techniques discussed during the thesis. Specifically in the sections dedicated to program slicing, regression testing, and design-by-contract verification. Finally, the chapter classifies and describes the main contributions done for each one of the previously mentioned areas, and describes the structure of the thesis.

2. The Program Slicing part is divided into five different chapters:

   - Chapter 2 introduces the basics of program slicing. This chapter describes how the *System Dependence Graph* (SDG) is built from the source code and how the standard program slicing algorithm obtains the slice by traversing it. Additionally, the chapter describes some key concepts of program slicing of object-oriented programs and alternative slicing techniques.

- Our new proposal for slicing object-oriented programs is explained in Chapter 3, where we point out some lacks of the current object-oriented program representation, the Java System Dependence Graph (JSysDG) when it is used to represent the dependences of object variables. In this chapter, we make an alternative and complementary proposal to accurately represent the dependences of object variables in object-oriented programs with its corresponding implementation and experimental evaluation. Most of the content of this chapter has been extracted from the article in [66], where some extra examples and extensions have been added.
- Chapter 4 presents two new program representation models for field-sensitive slicing, the Constrained-Edge Program Dependence Graph (CE-PDG) for intraprocedural slicing, and the Constrained-Edge System Dependence Graph (CE-SDG) for interprocedural slicing. The chapter details how to compute the CE-PDG from the PDG, how to join CE-PDGs to build the CE-SDG, and how to adapt the slicing algorithm to slice both graphs. Part of the content introduced in the chapter (the CE-PDG part) has been extracted from [61] while the CE-SDG part is original from this thesis.
- In Chapter 5, we propose a new fine-grained graph representation for programs, the *Expression Dependence Graph* (EDG), built from the AST of the program, where each literal of the source code is represented as a node of the graph. This chapter further elaborates the idea proposed in [63], introducing a set of situations where the SDG generates inaccurate slice, accurately slicing them with the use of the EDG.
- Finally, Chapter 6 proposes a method to compare program slicers with a benchmark suite based on quasi-minimal slices. The chapter presents the notion of quasi-minimal slice and it describes a methodology to compute them and to use them to compare the performance of different program slicers. Most of the content of this chapter has been extracted from [152].

3. The Testing and Verification part is divided into three different chapters:

   - Since the programming language analysed by the following chapters is Erlang, Chapter 7 shows the utility of different analysis tools for Erlang used in the following chapters in Section 7.1. Additionally, an overall description of the design-by-contract methodology is also provided by Section 7.2.
   - Chapter 8 describes a methodology to generate and execute a set of test cases to perform regression of an arbitrary program point in Erlang programs called Point Of Interest (POI) testing. The chapter

describes the internals of POI testing, illustrating the required program transformations to extract traces and runtime information, its applicability in programs with concurrency, and the implementation and evaluation of a tool to run POI testing in Erlang. The content of this chapter summarises the research line of POI testing, published in [87, 89, 153].

- Chapter 9 presents the implemented approach to apply design-by-contract verification in Erlang programs. In this chapter we describe each implemented contract and its associated syntax providing examples of how to use both sequential and concurrent available contracts in practice. Most of the material of this chapter has been extracted from [60], where some extra content about implementation details has been also included.

4. The Developed Tools part, composed by Chapter 10, details the functionality of all the implemented tools mentioned along this thesis. Each section of this chapter describes some features of each tool: installation requirements, installation process/commands, possible configurations, command line tool execution, and report analysis. Additionally, each tool section also includes some use cases with screenshots to guide the user in her first steps.

5. With respect to the Conclusions and future work part, it develops the conclusions of the thesis in Chapter 11 and the open lines of research that can be further explored in Chapter 12.

Part II

Program Slicing

Chapter 2

Preliminary Definitions and Notation

This chapter introduces the reader to the world of program slicing. Along the chapter we show the basics of program slicing, together with the graph representations and slicing variations used in different programming languages and paradigms. We show how the graph representation of the program is built, together with the evolution of this representation to deal with some program features like polymorphism and field representation in object-oriented programs, access to complex data structures such as lists or tuples in functional languages, and even some language-independent program slicing techniques that generate program slices by running the program with a set of test cases. We start with the formal definition of program slice, used as the base of all our research work.

Definition 2.1 (Program Slice [18]). *For statement s and variable v, the slice of program P with respect to the slicing criterion $\langle s, v\rangle$ includes only those statements of P needed to* capture *the behavior of v at s.*

An intuitive definition for "capture the behavior" is given in the definition of *executable slice*.

Definition 2.2 (Executable Program Slice [18]). *For statement s and variable v, the slice S of program P with respect to the slicing criterion $\langle s, v\rangle$ is any executable program with the following properties:*

1. *S can be obtained by deleting zero or more statements from P.*
2. *If P halts on input I, then the value of v at statement s each time s is executed in P is the same in P and S. If P fails to terminate normally, s may be executed more times in S than in P, but P and S compute the same values each time s is executed in P.*

As indicated in Definition 2.2, a code is an executable program slice if the *sequence of values* generated in the slicing criterion when executing the original program is a prefix of the sequence of values generated when executing the slice. When the sequence of values generated by the code computed by a program slicer does not fulfil the prefix property, it is not considered a slice, and is commonly denoted as "incomplete slice" (see Example 2.1). Thus, the terms *program slice* and *complete program slice* are indeed synonyms.

Definition 2.3 (Sequence of values)**.** *Let P be a program, C an slicing criterion of P, and I a possible input for P. $seq(P, C, I)$ represents the sequence of values the slicing criterion C is evaluated to during the execution of P with I.*

It is important to remark that we use the standard definition of slice, which excludes non-terminating and non-deterministic programs. Another important property is that we want our slices to be executable, so that the execution of the slice for any given input must evaluate the slicing criterion as many times (or more) as the original code, and the sequence of values the slicing criterion is evaluated to when executing the original code must be equal to (or a prefix of) the sequence obtained at the slice. Formally,

Definition 2.4 (Static executable program slice (based on [18] and [20]))**.** *A* static executable program slice *S of program P with respect to a slicing criterion C is any executable program with the following properties:*

1. *S can be obtained by deleting code from P (denoted $S \subseteq P$).*
2. *For all input I, $seq(P, C, I)$ is a prefix of $seq(S, C, I)$.*

As stated in Definition 2.2, a slice can be obtained by deleting zero statements from a program, so there must be an indicator to measure the quality of a program slice which must evaluate if a computed slice is valuable for its purpose or not. There is one indicator used along the literature that measure the quality of a computed program slice: A program slice is considered to be correct when all the statements included on it are strictly necessary to recreate the behaviour of the slicing criterion. Since a program slice is computed by deleting zero or more statements from a program, a program can contain many different slices for the same slicing criterion. From here on, given a program P and a slicing criterion C for P, we use the domain $\mathcal{S}lices^P_C$ to denote the finite set containing all possible slices of P with respect to C.

Definition 2.5 (Minimal slice)**.** *A minimal slice of program P with respect to a slicing criterion C is any $S \in \mathcal{S}lices^P_C$ such that $\nexists S' \in \mathcal{S}lices^P_C \wedge S' \subset S$.*

This definition states that a minimal slice is always a correct slice because none of its statements can be removed maintaining the original program behaviour for the slicing criterion C (see Example 2.1). Therefore, the terms minimal slice and correct slice can be used interchangeably. Note that a minimal slice, according to this definition, is not necessarily unique and is not necessarily a slice with the smallest number of statements (see, e.g., [48]).

Example 2.1 (Incomplete and minimal slices)**.** *Consider the program in Figure 2.1a which prints the odd numbers from 5 to 0 and computes their summation. Consider also two different slices computed w.r.t. $\langle 6, n\rangle$, with their associated execution traces for the slicing criterion. The slice in Figure 2.1b removes from the program the statement 8, necessary to replicate the same values for variable n, generating a slice that violates the prefix property mentioned in Definition 2.2, and required to be a slice. We say that the slice in Figure 2.1b is an "incomplete slice". On the other hand, the slice in Figure 2.1c contains*

```
1  int i = 0;
2  int n = 5;
3  int s = 0;
4  while(i < 5){
5    if (odd(n))
6      write(n);
7    s += n;
8    n--;
9    i++;
10 }
11 write(s);

   SEQ: 5, 3, 1
```

(A) Original program

```
int i = 0;
int n = 5;
int s = 0;
while(i < 5){
  if (odd(n))
    write(n);
  s += n;
  n--;
  i++;
}
write(s);

SEQ: 5, 5, 5, 5, 5
```

(B) Incomplete slice

```
int i = 0;
int n = 5;
int s = 0;
while(i < 5){
  if (odd(n))
    write(n);
  s += n;
  n--;
  i++;
}
write(s);

SEQ: 5, 3, 1
```

(C) Minimal/Correct slice

FIGURE 2.1: Example of incomplete and minimal slices w.r.t. $\langle 6, n\rangle$

all the statements required to compute, for variable n, the same values that the original program. Furthermore, if any of the statements in the slice was removed, the new computed sequence of values would differ from the original and the code would no longer be a slice. We say that the slice in Figure 2.1c is a minimal (or correct) slice.

The computation of program slices is not a trivial process. Along the literature, two main approaches are proposed to compute program slices: the approach based on data flow equations, and the one based on graph reachability problems [195]. In the data flow equations approach, the slice is computed in an iterative process, by computing sets of relevant variables in the CFG. These sets are classified into directly relevant variables (variables defined in a CFG node and later referenced in another CFG node) and indirectly relevant variables (those variables used in predicates that may affect the execution of a CFG node). When a CFG node is included in the slice, its variable sets are computed, including the CFG nodes that contain these variables in the slice. The process repeats until a fixpoint is reached, obtaining the desired program slice. On the other hand, graph reachability approaches represent the whole program as a graph, which includes a set of dependences between program statements computed from the CFG. Then, an algorithm traverses the graph starting from the slicing criterion, including in the slice all the graph reachable nodes. All the contributions of this thesis have been done using the graph reachability program slicing approach, including enhancements to the graph representation of programs and adapting slicing algorithms to work with them. The rest of the chapter describes how the SDG is built from the source code and how the standard program slicing algorithm obtains the slice by traversing it in Sections 2.1 and 2.2. Then, Section 2.3 shows how the SDG is improved to the JSysDG to represent object-oriented programs and Section 2.4 describes another language-independent slicing method, observation-based slicing, later used in one of the proposed techniques.

2.1 Program Slicing with System Dependence Graphs

Since it was defined in 1988 by Horwitz et al. [83], the System Dependence Graph (SDG) is at the basis of most program slicing techniques. The SDG is defined from the CFG. We explain it through its incremental evolution:

$$CFG \rightarrow PDG \rightarrow SDG$$

CFG. The starting graph to build a SDG is the CFG [5]. The CFG is a graph that represents all possible execution paths of a routine (function/procedure/method...). In the CFG each statement is represented with a node, and two nodes are connected if there exists a possible control flow between these statements. Additionally, two nodes, *Enter* and *Exit*, are added as the initial and final nodes of the method execution respectively. Formally,

Definition 2.6 (Control Flow Graph (based on [5]))**.** *Given a method M, which contains a set of statements R, the* control flow graph *of M is a directed graph $G = (N, E)$, where $N = R \cup \{\text{Enter}, \text{Exit}\}$ and E is a set of edges of the form $e = (n, m)\,|n, m \in N$ where the statement represented by m may be executed immediately after the statement represented by n.*

Each CFG node includes information about the variables defined and used in the statement it represents. This information is stored in two different sets associated to each CFG node: the *DEF* set (for defined variables) and the *USE* set (for used variables).

PDG. The first program representation used to define program dependences for program slicing was the *Program Dependence Graph* (PDG), defined by Ferrante et al. in [56]. The PDG defines each program method as an individual graph, where nodes represent statements, and edges connect statements with dependence arcs. These dependences are the *control dependence* and the *flow dependence*, and they are defined hereunder.

Definition 2.7 (Control dependence)**.** *Let G be a CFG. Let n and m be nodes in G. A node n is post-dominated by a node m in G if every directed path from n to the Exit node passes through m. Node m is* control dependent *on node n if and only if m post-dominates one but not all of n's CFG successors.*

Definition 2.8 (Flow Dependence)**.** *A node m is* flow dependent *on a preceding node n if:*

(i) n defines a variable v,

(ii) m uses v, and

(iii) there exists a control-flow path from n to m where v is not defined.

The PDG of a procedure is a graph $G = (N, E)$ where N is the set of nodes of the CFG without the *Exit* node, and E is a set of edges that represent control and flow dependences. Formally,

Definition 2.9 (Program dependence graph)**.** *Given a method M and its associated CFG $G = (N, E)$, the PDG of M is a directed graph $G' = (N', E')$, where $N' = N \setminus \{Exit\}$ and $E' = E_c \cup E_f$, being E_c and E_f the set of edges computed by definitions 2.7 and 2.8 over G, respectively.*

SDG. A program usually contains a set of procedures connected by procedure calls. For this reason, in order to connect all the procedures of a program in a single graph, the SDG was defined [85]. A SDG is compositionally constructed by connecting the PDGs of program procedures to model parameter passing between procedure calls and procedure definitions. The SDG represents each parameter of a procedure with a formal-in node, and a formal-out node for each entry parameter that may be modified inside the procedure. Analogously, each procedure call is augmented with an actual-in node for each argument of the call, and an actual-out node for each argument that may be modified inside the procedure. The SDG connects procedure calls with procedure definitions representing parameter passing by means of parameter edges: parameter-in edges connect actual-in with formal-in nodes and parameter-out edges connect formal-out with actual-out nodes. Additionally, a call edge is generated to connect the call node to the *Enter* procedure node. Finally, a new kind of edge called summary edge is added to the SDG to describe the relationship between defined and used arguments in method calls. A summary edge connects an actual-in node and an actual-out node if the value related to the actual-in node is used to calculate the value defined in the actual-out node. All this representation can be seen in the example shown in Figure 2.2, where parameter-in and parameter-out edges are called input and output edges respectively, name that we will use from here on.

Definition 2.10 (System dependence graph)**.** *Given a program P, composed of a set of methods $M = \{m_0 ... m_n\}$ and their associated PDGs—each method m_i has a $PDG^i = \langle N^i, E^i_c, E^i_d \rangle$. The SDG of P is a graph $G = \langle N, E_c, E_d, E_{call}, E_{in}, E_{out}, E_{sum} \rangle$ where:*

1. *$N = \bigcup_{i=0}^{n} N^i$ (from now on, we will consider $n, m \in N$)*
2. *$E_c = \bigcup_{i=0}^{n} E^i_c$*
3. *$E_d = \bigcup_{i=0}^{n} E^i_d$*
4. *$(n, m) \in E_{call}$ if and only if n is a statement that contains a call and m is an "Enter" node of the method called by n. (n, m) is a call edge.*
5. *$(n, m) \in E_{in}$ if and only if n is an actual-in node of a call node c, m is a formal-in node of an "Enter" node e, and $(c, e) \in E_{call}$. (n, m) is a parameter-in edge.*
6. *$(n, m) \in E_{out}$ if and only if n is a formal-out node of an "Enter" node e, m is an actual-out node of a call node c, and $(c, e) \in E_{call}$. (n, m) is a parameter-out edge.*

7. *$(n, m) \in E_{sum}$ if and only if n, m are actual-in and actual-out nodes of a call node c, respectively, n', m' are formal-in and formal-out nodes of a "Enter" node e, respectively, $(c, e) \in E_{call}$, and there is a path from n' to m'. (n, m) is a summary edge.*

```
1   main() {
2     sum = 0;
3     i = 1;
4     while (i < 11) {
5       sum = add(sum, i);
6       i = increment(i);
7     }
8     write(sum);
9   }

10  add(a,b) {
11    return a + b;
12  }

13  increment(x) {
14    return add(x,1);
15  }
```

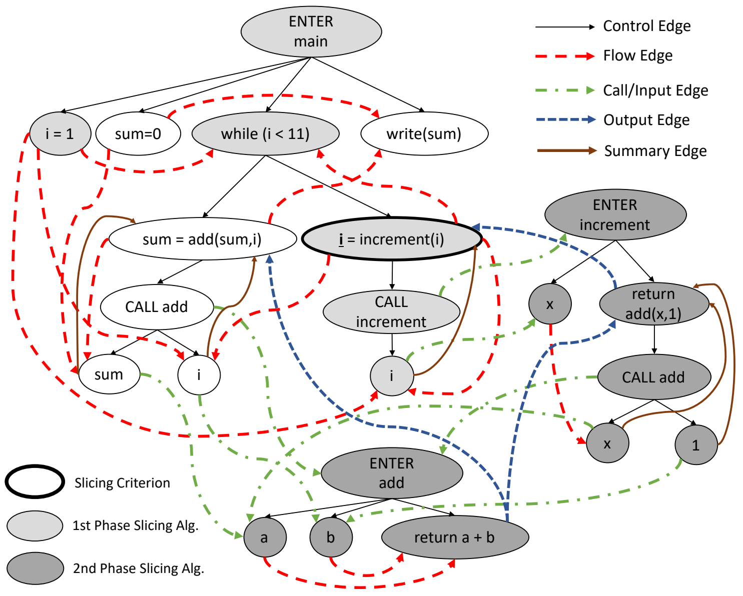


FIGURE 2.2: Program extracted from [85], its associated SDG, and the slice for $\langle 6, i \rangle$, represented with grey nodes

2.2 Program Slicing Algorithm

The program slicing algorithm to compute program slices by traversing the SDG proposed by Horwitz et al. in [85] is illustrated in Algorithm 2.1. In Algorithm 2.1, the computation of the slice in the SDG (function MARKNODESOFSLICE) is divided into two phases (lines 2 and 3). Both phases traverse all the control and flow edges backwards to reach all possible nodes based on these dependences, but each phase ignores a specific kind of edges (parameter *EdgeTypes* in function MARKREACHINGNODES) with a particular purpose:

- **Phase 1** identifies nodes that can reach the slicing criterion, and are either in the same method as the slicing criterion itself (M), or in a method that directly or transitively calls M. Since output edges are not traversed (call in line 2), phase 1 does not include in the slice the methods called by M.

- **Phase 2** locates nodes that can reach the slicing criterion from methods called by M or from methods called by methods that transitively call M. Since call and input edges are not traversed in this phase (call in line 3), phase 2 only includes in the slice the slicing criterion context that calls this methods, ignoring all possible calls along the program. This fact makes the algorithm proposed by Horwitz et al. to be aware of the calling contexts, also known as being *context-sensitive*.

The graph traversal process is performed with function MarkReachingNodes. This function receives a graph G, a set of nodes N that must be in the slice, and the set of edge types ignored during the traversal *EdgeTypes*. The function uses an auxiliary work list (*WorkList*) to store the nodes in N (line 6). For each node, the function extracts all its incoming edges and traverses those which type is not in *EdgeTypes* (lines 8-11). If the edge can be traversed, its source node is included to the slice and also to the work list, to be later processed (lines 12-14). This process repeats until no more nodes are added to the work list.

Algorithm 2.1 Program Slicing Algorithm defined by Horwitz et al. in [85]

Input: The SDG G and the slicing criterion node n_{sc}.
Output: The set of nodes that compose the slice S.
Initialization: $S_0 = \{n_{sc}\}$.

```
 1: function MarkNodesOfSlice($G, n_{sc}$)
      // Phase 1: Ignore Output edges
 2:   $S_1 \leftarrow$ MarkReachingNodes($G, S_0, \{Output\}$)
      // Phase 2: Ignore Call and Input edges
 3:   $S \leftarrow$ MarkReachingNodes($G, S_1, \{Call, Input\}$)
 4:   return $S$

 5: function MarkReachingNodes($G, N, EdgeTypes$)
 6:   $WorkList \leftarrow N$
 7:   while $WorkList \neq \emptyset$ do
 8:     select some $n \in WorkList$
 9:     $WorkList \leftarrow WorkList \setminus \{n\}$
10:     for all $edge \in$ getIncomingEdges($n$) do
11:       if $edgeType \notin EdgeTypes$ then
12:         $m \leftarrow$ getSourceNode($edge$)
13:         $N \leftarrow N \cup m$
14:         $WorkList \leftarrow WorkList \cup m$
15:   return $N$
```

Figure 2.2 shows a program, its associated SDG, and the interprocedural slicing (marked with grey nodes) for the slicing criterion $\langle 6, i\rangle$ (marked in underlined bold inside a bold node). The graph traversal performed by the slicing algorithm in each phase would be the following:

Phase 1:

$$\begin{tikzcd}
 & \texttt{i} \arrow[dl, "summary"'] & \texttt{CALL increment} \arrow[l, "control"'] \\
\texttt{i = increment(i)} & \texttt{while (i < 11)} \arrow[l, "control"'] & \texttt{ENTER main} \arrow[l, "control"'] \\
 & & \texttt{i=1} \arrow[ul, "flow"]
\end{tikzcd}$$

Phase 2:

$$\texttt{i = increment(i)} \xleftarrow{\mathit{output}} \texttt{return add(x,1)} \left\{\begin{array}{ll} \xleftarrow{\mathit{summary}} & \texttt{1} \\ \xleftarrow{\mathit{summary}} & \texttt{x} \xleftarrow{\mathit{flow}} \texttt{x} \\ \xleftarrow{\mathit{control}} & \texttt{ENTER increment} \\ \xleftarrow{\mathit{output}} & \texttt{return a + b} \left\{\begin{array}{ll} \xleftarrow{\mathit{control}} & \texttt{ENTER add} \\ \xleftarrow{\mathit{flow}} & \texttt{a} \\ \xleftarrow{\mathit{flow}} & \texttt{b} \end{array}\right. \end{array}\right.$$

where arrows represent the graph edges, which connect program statements; and their labels represent the type of the edge. Note that we use braces with a collection of edges when more than one edge can be traversed, thus, creating parallel paths. In Phase 1, output edges are ignored and, thus, only some nodes inside the `main` function are included in the slice. In Phase 2, the traversal reaches the `increment` and `add` functions (traversing the output edges). Moreover, since in Phase 2 call and input edges are ignored, the traversal never reaches the call to function `add` performed inside the `main` function.

2.3 Program Slicing of Object-Oriented Programs

Many approaches have been proposed to enhance the SDG and make it suitable to represent object-oriented (OO) programs (most of these approaches are explained in the survey [136]). To deal with OO programs, the SDG needs to provide an accurate representation for their main features: polymorphism, dynamic binding, and inheritance.

One of the latest and more accurate representation for Java OO programs, is the one proposed by Walkinshaw et al. [202], the Java System Dependence Graph (JSysDG). The JSysDG is made by a composition of different graphs for methods, classes, interfaces, and packages, obtaining a more accurate program representation. The JSysDG allows the representation of abstract classes which are not necessarily interfaces, and it distinguishes data members (the fields of an object) in objects passed as parameters in method calls. Nowadays, the JSysDG is arguably the best representation for Java OO programs and we use it as a reference point in this thesis.

In order to clarify the differences between the SDG and the JSysDG, we explain the JSysDG enhancements introduced to correctly represent some OO features such as inheritance, dynamic binding, and polymorphism. We explain it through its incremental evolution:

$$\mathit{SDG} \rightarrow \mathit{ClDG} \rightarrow \mathit{JSysDG}$$

ClDG. With the SDG as its base, the *Class Dependence Graph* (ClDG) [112] augments the SDG representation to consider OO programs. The ClDG presents a representation for polymorphism, dynamic binding, and inheritance in the C++ programming language. Unfortunately, the representation is not accurate enough and presents some limitations. For example, it is not possible to differentiate data members of different objects in method calls. Liang and Harrold

[118] improved this representation allowing them to distinguish data members in parameter objects and upgrading the accuracy of graph-based operations as a result. As some features differ among OO languages, the ClDG is not able to represent some features of the Java programming language. Hence, other approaches focused on Java have been proposed. The approach proposed by Kovács et al. in [101] or the one proposed by Zhao in [215] enable the representation of Java particular features such as interfaces, packages, and single inheritance. All these ClDG approaches define a *class entry node* for each class, connected to the procedure *Enter* nodes of all its procedures by *class membership edges*, and to all its data members by *data membership edges*. Additionally, inheritance is represented with a *class dependence edge* from the base class to the derived classes.

Definition 2.11 (Class dependence graph)**.** *Given a program P, composed by a set of classes $Cl = \{cl_0 ... cl_n\}$ that contain a set of methods and a list of data members DM_{cl_i} and the SDG associated to P, $G = \langle N, E\rangle$; the ClDG of P is a graph $G' = \langle N', E, E_{cm}, E_{dm}, E_{cd}\rangle$ where:*

1. *$N' = N \cup Cl \cup \bigcup_{i=0}^{n} DM_{cl_i}$*
2. *$(n, m) \in E_{cm}$ if and only if $n \in Cl$ and m is an "Enter" node of a method defined inside class n. (n, m) is a class membership edge.*
3. *$(n, m) \in E_{dm}$ if and only if $n \in Cl$ and $m \in DM_n$. (n, m) is a data membership edge.*
4. *$(n, m) \in E_{cd}$ if and only if $n, m \in Cl$ and m is a derived class of n. (n, m) is a class dependence edge.*

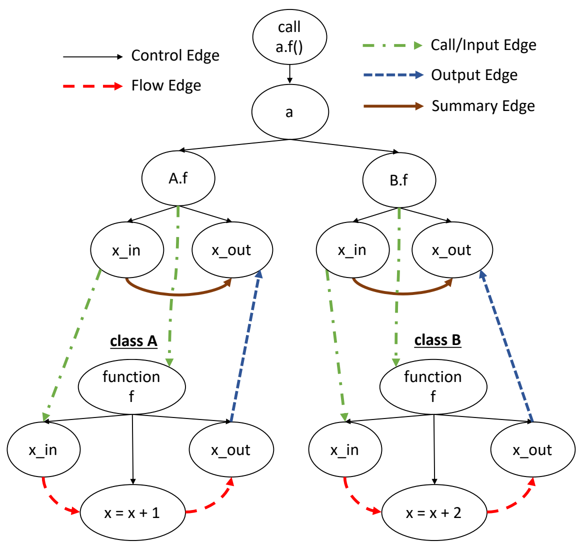
JSysDG. Taking profit of some features during the evolution of the ClDG, Walkinshaw et al. [202] proposed the JSysDG. This graph augments the ClDG with a process to represent polymorphic calls and dynamic binding in Java OO programs. There are two particular scenarios that are worth mentioning in the JSysDG representation.

1. **A polymorphic object is the caller of a method, and the called method needs to be selected at runtime**. An example of this scenario can be seen in the code of Figure 2.3a, which represents a Java program with two classes, `A` and `B`, with an inheritance relationship and a class `Main` with a `main` method that creates an object of either `A` or `B` dynamic type depending on a random generated number. In line 19, the method that would be executed in the call `a.f()` can only be determined at runtime, thus, the static representation of the program needs to define both possibilities. In this scenario, the JSysDG represents the caller (object variable `a`) and its defined and used data members in a tree representation for all the possible dynamic types and connects each type with its corresponding method definition. An example of this representation is shown in Figure 2.3b.

```
1   class A{
2     public int x, y;
3     public A (int a, int b) { x = a; y = b; }
4     public int getX() { return x; }
5     public int getY() { return y; }
6     public void f() { x = x + 1; }
7   }
8   class B extends A{
9     public B (int a, int b){ super(a,b); }
10    public void f() { x = x + 2; }
11  }

12  class Main{
13    public static void main(String[] args){
14      A a;
15      if (Math.random() > 0.5)
16        a = new A(1);
17      else
18        a = new B(2);
19      a.f();
20      g(a);
21    }
22    public void g(A a){
23      System.out.println(a.getX() + a.getY());
24    }
25  }
```

(A) Java program with polymorphic calls



(B) JSysDG of call `a.f()` in line 19

FIGURE 2.3: Fragment of Java code of a polymorphic call and JSysDG representation for method call `a.f()`

2. **A method call contains a polymorphic object as a call parameter.** This scenario happens in line 20 of Figure 2.3a. The call to method `g` receives a polymorphic object (variable `a`) as a parameter. In this scenario the JSysDG representation follows the proposal introduced by Liang and Harrold in [118], where the object parameter has a tree representation for each class, unfolding all its data members in both method call and definition. An example of this representation for the call `g(a)` can be seen in Figure 2.4. In the proposed representation, only data members of the tree representation are linked, in order to accurately select a data member if it is used by method `g`.

Apart from the augmented ClDG, the JSysDG also uses two other graphs to account for interfaces and packages. The first graph is the *Interface Dependence Graph* (InDG), which represents each interface of the program with an interface entry node. This node is connected to every abstract method defined inside the interface. Each abstract method contains in turn a set of parameter nodes, which represent its input parameters. Then, every abstract method and its parameters are connected with an *implement abstract method edge* to every instance of the method. Finally, if a class implements an interface, an *implements interface edge* from the interface node to the class node is added to the graph.

The second graph is the *Package Dependence Graph* (PaDG). In this graph, a *package entry node* is defined for each package of the program. This node is connected to every class and interface inside the package with a *package member*

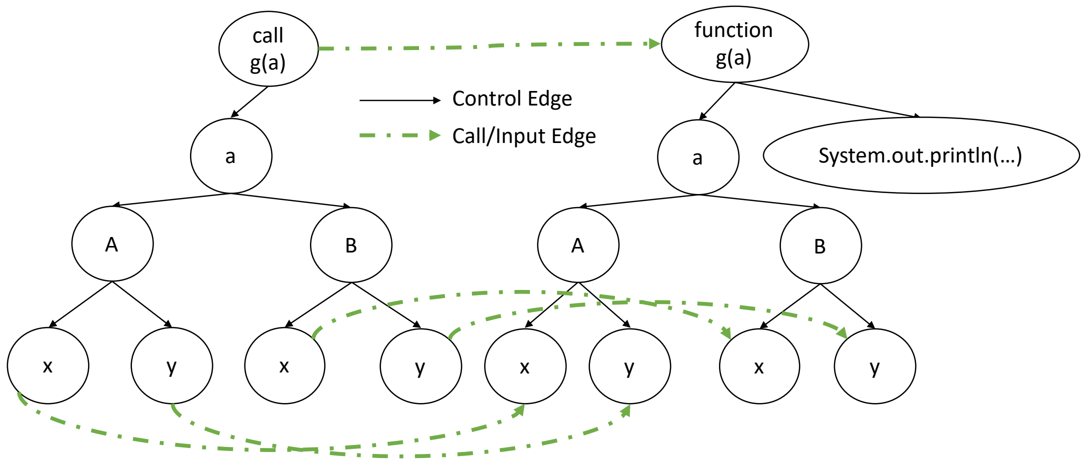


FIGURE 2.4: JSysDG of call `g(a)` in line 20 of the code in Figure 2.3a

edge. This graph, which includes all the previous graphs, represents the Java program as a whole, establishing dependences at package level.

2.4 Language-Independent Program Slicing: Observation-Based Slicing

Although all the approaches explained until now were based on program dependence graphs, there are other slicing approaches that do not consider program dependences or program representations to compute slices. This is the case of *observation-based slicing* (ORBS) [20]. ORBS is a technique that iteratively removes lines from a program, and checks whether the observable behaviour of the slicing criterion is the desired one. This is checked for a particular set of test cases. If the observable behaviour is the expected one, then the line is effectively removed. Then, the system can try again with a different line until no more lines can be removed. When the system has finished with one line at a time, it can repeat the process removing two lines at each iteration, and so on. Since the removal of lines is based on the behaviour of the program execution, this strategy requires a good defined set of test cases that covers all possible executions to be effective.

In this section we formalise some key definitions that must be used when dealing with program slicing using ORBS. Since ORBS makes use of program execution to compute the slice it can be considered as a dynamic technique. In this case, the concept of slicing criterion must be augmented to the concept of *dynamic slicing criterion.*

Definition 2.12 (Dynamic slicing criterion)**.** *Let P be a program. A dynamic slicing criterion of P is a tuple $\langle C, I\rangle$ such that C is a slicing criterion and I is an input for P.*

A dynamic slicing criterion also includes extra information: the input of the program. In the slice, the behaviour of the slicing criterion when executing the program with this input information must remain. To consider if the behaviour is preserved or not, the process must monitor the sequence of values (Definition 2.3) computed for the slicing criterion during the program execution. The dynamic requirements of ORBS and the definition of the dynamic slicing criteria make the slices computed by ORBS to be dynamic executable program slices.

Definition 2.13 (Dynamic executable program slice)**.** *A* dynamic executable program slice *S of a program P on a dynamic slicing criterion $\langle C, I\rangle$ is any executable program that fulfils the two properties of Definition 2.2 for P with respect to C and for a set of possible inputs I:*

1. *S can be obtained by deleting zero or more statements from P.*

2. *If P halts on input I, then the value computed for C each time it is executed in P is the same in P and S. If P fails to terminate normally*[1]*, C may be executed more times in S than in P, but P and S compute the same values each time C is executed in P.*

These terms and definitions are valuable to understand ORBS and its dynamic context, and will be used in future chapters.

[1]We consider that a program does not terminate normally when an uncontrolled exception is raised during runtime and the program's execution is abruptly interrupted.

Chapter 3

Flow Dependence for Java Object-Oriented Programs

The JSysDG (Chapter 2, Section 2.3) is an accurate representation that provides precise slices and allows for differentiating whether data members of different objects are required or not in a slice. In this chapter, we show that, despite being one of the most accurate approaches for OO programs, some scenarios exist where the produced slices are not complete (Section 3.1). Therefore, it is not a problem related to precision (the slices contain more code than they need), but a problem related to completeness (the slices contain less code than they need), which means that some code that can affect the slicing criterion is not included in the slice. In particular, when an object variable is selected as the slicing criterion, under certain circumstances, only some of its required data members (not all of them) are included as part of the slice. The cause of this lack of completeness is the current definition of flow dependence. The classic flow dependence definition was thought for variables that are atomically defined or used in a program statement, but has never been reconsidered to deal with object variables, which can be partially defined or used (by defining or using only one of its data members) in a statement. This representation error was inherited by later program representations derived from the JSysDG, such as the Sub-Statement Level Dependence Graph (SSLDG)[128].

This chapter presents an approach to solve the expressiveness limitation by extending the JSysDG. We augment the JSysDG by replacing the current definition of flow dependence with two more accurate definitions: a definition of flow dependence for primitive type variables, and the creation of two new definitions related to flow dependence for object variables: object-flow dependence and object-reference dependence (Section 3.2). Finally, we show the implementation and experimental evaluation of our proposal in a program slicer called JavaSlicer (Sections 3.3 and 3.4).

3.1 Limitations of the JSysDG

According to the definition of program slice (see Definition 2.1), there is nothing that prevents an object type variable to be considered as the slicing criterion. Moreover, if we select an object variable as the slicing criterion, one could expect for the slice to include all the statements that might affect the value of any of its data members. Nevertheless, as we have seen in Figure 2.3b, when an object variable is the caller of a method call, only the used and defined data members of the object have a representation in the unfolding tree as argument-in and argument-out nodes respectively. Additionally, we have seen that all the flow dependences between object definitions and uses in method calls are propagated through their data members, never connecting the node that represents the object variable itself.

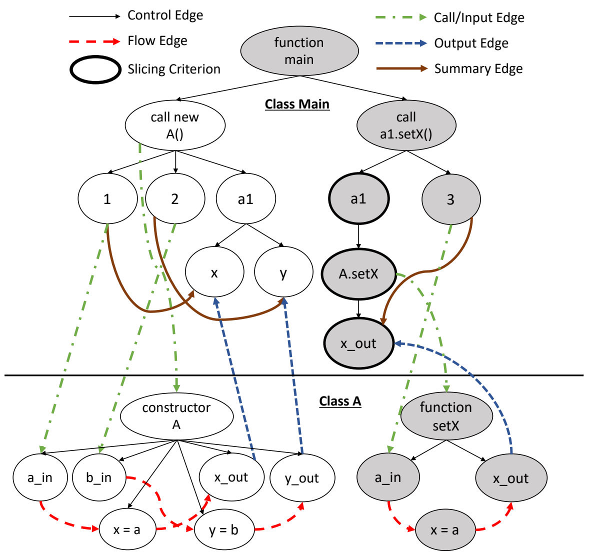
When we put all this knowledge together, we can easily find a specific scenario where the slice obtained by the JSysDG is not complete when selecting some object variables as the slicing criterion.

Example 3.1. *Consider the code snippet in Figure 3.1a. This code contains a program with two classes: class `A` and class `Main`. Additionally, class `Main` contains a method `main` that instantiates object variable `a1` of type `A`. The scenario where the JSysDG is incomplete is given at the `main` method of Figure 3.1a, concretely in lines 11 and 13. Line 11 creates the object variable `a1` of type `A` instantiating its data members `x` and `y` with a call to `A`'s constructor. Then, line 13 modifies only the value of its data member `x`. At this point, after the method call `a1.setX()`, the whole definition of the object is divided into two different statements, but there is no flow dependence between them, because method call `a1.setX()` does not require any data member of object variable `a1` to define data member `x`. The described situation is represented in Figure 3.1b.*

```
 1  class A {
 2    public int x,y;
 3    public A (int a, int b) {
 4      x = a;
 5      y = b;
 6    }
 7    public void setX(int a) { x = a; }
 8  }

 9  class Main {
10    public static void main(String[] args) {
11      A a1 = new A(1,2);
12      a1.y = a1.y + 5;
13      a1.setX(3);
14      A a2 = a1;
15    }
16  }
```

(A) Java program



(B) JSysDG representing lines 11 and 13 in method `main`, and slice w.r.t. ⟨13, `a1`⟩

FIGURE 3.1: Java program, JSysDG of lines 11 and 13, and slice w.r.t. ⟨13, `a1`⟩

Figure 3.1b represents with grey nodes the slice produced by the JSysDG with respect to ⟨13, `a1`⟩. This slicing criterion is marked in the JSysDG with bold nodes. Note that, when the slicing criterion is an object variable, then all the data members are marked as the slicing criterion as well. Thus, the slicing criterion is formed from the control dependence subtree of node `a1`. The translation of the slice calculated over the JSysDG to code results in the code shown *in Figure 3.2a. Contrarily to the expected slice, which is the one in Figure 3.2b, the slicing traversal of the JSysDG ends without reaching the declaration of* `a1`*, the constructor call* `A(1,2)`*, and data member* `y` *in class* `A`*. This counterexample reinforces the claim that flow dependences of object variables need to be better defined to correctly use them as slicing criteria.*

```
 1  class A {
 2    public int x,y;
 3    public A (int a, int b) {
 4      x = a;
 5      y = b;
 6    }
 7    public void setX(int a) { x = a; }
 8  }

 9  class Main {
10    public static void main(String[] args) {
11      A a1 = new A(1,2);
12      a1.y = a1.y + 5;
13      a1.setX(3);
14      A a2 = a1;
15    }
16  }
```

(A) JSysDG slice

```
 1  class A {
 2    public int x,y;
 3    public A (int a, int b) {
 4      x = a;
 5      y = b;
 6    }
 7    public void setX(int a) { x = a; }
 8  }

 9  class Main {
10    public static void main(String[] args) {
11      A a1 = new A(1,2);
12      a1.y = a1.y + 5;
13      a1.setX(3);
14      A a2 = a1;
15    }
16  }
```

(B) Expected slice

FIGURE 3.2: JSysDG slice and expected slice of the code in Figure 3.1a w.r.t. $\langle 13, \texttt{a1} \rangle$

3.2 A novel definition of flow dependence

A more accurate description for definition and use sets of object variables is necessary in order to extend the notion of flow dependence (see Definition 2.8) for objects due to the different natures of primitives and object variables.

In Java, program variables can be of two different types. On the one hand, we have primitive type variables, which are atomic (e.g., int i = 42). These variables are always defined and used atomically, i.e., every time a primitive variable is defined in the program, the new value of the variable replaces the previous one, and the previous value cannot be further accessed. On the other hand, there are object type variables, which are compositionally formed by a collection of data members. Each data member, in turn, can be a primitive type variable, or another object type variable. Unlike primitive variables, object variables are not completely replaced every time they are defined. Since they are formed from a collection of data members, a statement may modify just some of them. As a result, the definition of all the data members of an object variable may be split into different statements. Hence, object variables can be defined in two different ways:

Definition 3.1 (Total definition)**.** *An object variable v that points to a memory location $m1$ is* totally defined *in a program statement s if the execution of s makes v to point to $m2$, and $m1 \neq m2$ (v points to a different object).*

Total definitions appear in two different scenarios:

1. **An assignment of an object variable with a constructor call.** This happens, e.g., in `A a1 = new A(1,2)` (see line 11 of Figure 3.1a), where `a1` is totally defined. Constructor methods always define[1] all the data members of the variable and never use them. Hence, the *DEF* set (see *DEF* and *USE* sets of the CFG in Section 2.1) for this statement ($DEF(\#11)$) includes the object variable itself and all its data members. The order in which all these definitions occur is crucial for the definition of flow dependence. For this reason, we transform the *DEF* set into an ordered sequence called DEF_{os} where all data members are defined first, and the object variable is defined at the end, e.g., the ordered sequence of definitions in the mentioned statement is $DEF_{os}(\#11)$ = [`a1.x, a1.y, a1`]. In this scenario, $USE(\#11) = \emptyset$ since no variable is used.

2. **An alias assignment between two object variables that point to different objects.** This happens, e.g., in `A a2 = a1` (see line 14 of Figure 3.1a), where `a2` is totally defined[2]. As in the previous scenario, the *DEF* set is transformed into an ordered sequence where data members are defined first, and the object variable is defined at the end. In contrast, $USE(\#14)$ remains as a set of uses, and contains the object variable of the right-hand side of the assignment ({`a1`}).

Definition 3.2 (Partial definition). *An object variable v is* partially defined *in a program statement s if v points to the same object o before and after the execution of s, and s defines at least one data member of o.*

Partial definitions occur when a statement modifies at least one data member of an object variable, but not the object it points to. Partial definitions appear in two different scenarios:

3. **A method call that defines a data member of the caller.** This happens, e.g., in `a1.setX(3)` (see line 13 of Figure 3.1a). Method calls may partially define object variables through method calls by defining some or all its the data members. In this scenario, the *DEF* set is also transformed to an ordered sequence which, as it happened in total definitions, includes all the data members defined inside the method followed by the object variable itself. On the other hand, the *USE* set includes all the data members used inside the method together with the caller object itself, whose reference is needed to make the call.

4. **An assignment of an object variable's data member.** This happens, e.g., in `a1.y = a1.y + 5` (see line 12 of Figure 3.1a). This scenario behaves in a similar way that the previous one. The *DEF* set is also transformed into an ordered sequence that contains an element for defined data members (`a1.y` in this case) followed by the object variable itself (`a1`). The *USE* set contains the data members (`a1.y`) and the object variables used (`a1`).

[1]This definition can be explicit or implicit, because Java initializes all data members by default.

[2]This type of total definitions include also assignments with the form `A a = f();` where an object variable is defined through the return of a method call.

Example 3.2. *Table 3.1 provides an example of DEF and USE sets for the* `main` *method of the program in Figure 3.1a. In this table there are examples of the described scenarios. Definitions and uses appear in the order described in the previous scenarios.*

Scenario	Statement	DEF_{os}	*USE*
1	`A a1 = new A(1,2);`	`[a1.x, a1.y, a1]`	$\emptyset$
4	`a1.y = a1.y + 5;`	`[a1.y, a1]`	`{a1.y, a1}`
3	`a1.setX(3);`	`[a1.x, a1]`	`{a1}`
2	`A a2 = a1;`	`[a2.x, a2.y, a2]`	`{a1}`

TABLE 3.1: The ordered sequence DEF_{os} and the *USE* set of some of the statements in the `main` method of Figure 3.1a.

This new method to annotate definitions and uses of object variables in program statements is of fundamental importance for the redefinition of flow dependence. Moreover, it comes with an important advantage over the previous formulation: It allows us to differentiate the specific moment in which a data member and an object is defined or used and, thus, it allows us to slice a program with respect to a specific instant in the computation (for instance, we can slice `a.setX(3)` with respect to the value of `a` 'before' or 'after' the method invocation). To incorporate this feature to the JSysDG, we have enhanced the graph representation for object variables at the scope of a method call. On the one hand, we provide an object tree for the scope to represent its value before the call (*scope_in*). The root of this tree represents the use of the object variable, and the tree contains the data members used inside the function call. On the other hand, if the object is modified during the method invocation, we provide another object tree that represents its value after the call (*scope_out*). The root of this second tree represents the definition of the object variable after the execution of the method and it contains the data members defined inside the call. For instance, considering the method call `a.setX(3)`, the resulting JSysDG would be the one shown in Figure 3.3.

New representation for explicit data member accesses

Direct definitions or uses of public data members outside their defining classes is an uncommon programming practice and is usually considered unsafe. Until now, most program representations approaches (including the JSysDG) do not provide any specific representation for this programming mechanism. Instead, it has traditionally been a common practice to replace direct accesses to public data members by the corresponding *set* and *get* method calls in a pre-processing step before building the graph [118]. In this case our approach is in line with the one proposed in SSLDG: the unfolding of the objects being directly defined/used in the corresponding statements. In our case, a tree representation is given for each defined/used object and data member in the statement making the direct access, in the same way it is done when they are defined/used through method calls. Additionally, when the definition of a data member depends on a data

```
   // Class A
public void setX(int a) { x = a; }

   // Class Main
public static void main(String[] args){
   a1.setX(3);
}
```

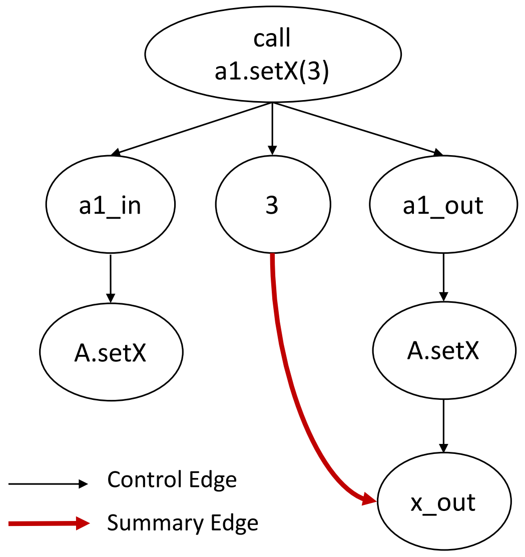


FIGURE 3.3: Input and output scopes in a method call

member used in the same statement, a flow edge between both data members represents the corresponding intra-statement flow dependence.

Example 3.3. *Consider the data member assignment in line 12 of Figure 3.1a. In the assignment, data member* `y` *of object variable* `a1` *is both used and defined. The representation of this assignment is shown in Figure 3.4. Since the same object is both used and defined, we represent the node with two different object trees: one corresponding to the value of the data member* `y` *of the right-hand side of the assignment (*`a1_in`*), and another one corresponding to the value of the data member* `y` *of the left-hand side of the assignment (*`a1_out`*). Finally, since the previous value of* `a1.y` *is used to compute the new value of* `a1.y`*, a flow edge is added to represent this dependence.*

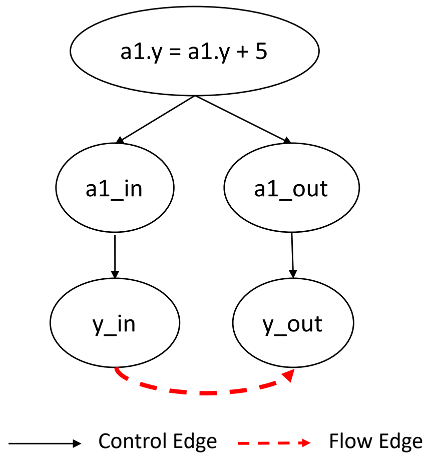


FIGURE 3.4: Representation for explicit data members accesses

Flow dependences for object variables

In Java, as it happens in most programming languages, a variable cannot be used without being previously defined. The definition of a primitive variable is always total, but the definition of an object variable can be partial. This poses new difficulties in the computation of the usual flow dependence. In fact, the standard definition (see Definition 2.8) is insufficient. Let us illustrate the problem with a simple example.

Example 3.4. *Consider the code in Figure 3.5a, where primitive variable* `x` *is defined twice and then used. According to Definition 2.8,* `x` *in line 4 depends on* `x` *in line 3, but* `x` *in line 4 does not depend on* `x` *in line 2.*

```
1        (...)
2        int x = 5;    // def x
3        x = 1;        // def x
4        y = x;        // use x
```

(A) Def-use with primitive variables.

```
1        (...)
2        A a = new A();    // def a
3        a.setX(1);        // def a
4        a.y = 2;          // def a
5        b = a;            // use a
```

(B) Def-use with object variables.

FIGURE 3.5: Flow dependence in presence of primitive and object variables

If we consider, however, the code in Figure 3.5b, we have an analogous situation: `a` *is defined twice and then used. Therefore, according to the standard definition of flow dependence,* `a` *in line 5 flow depends on* `a` *in line 4 (which is correct), but* `a` *in line 5 does not depend on* `a` *in line 3 (which is clearly wrong).*

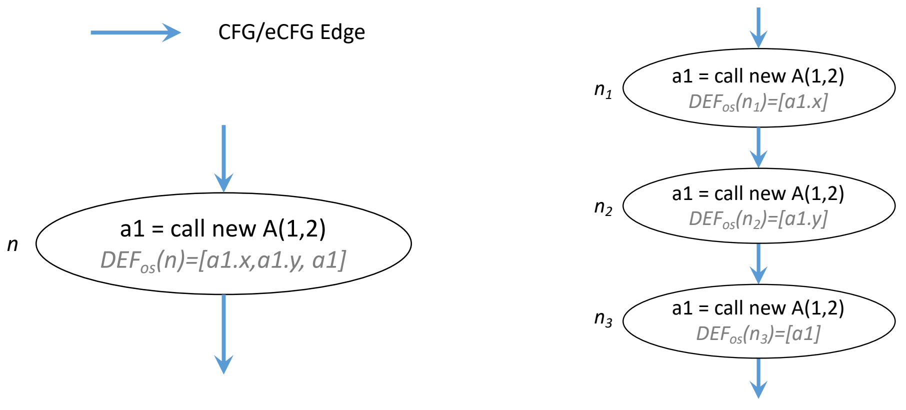
The rationale why Definition 2.8 does not work with object variables is because they can be *partially* defined, while primitive variables are always *totally* defined. This is also the problem of the JSysDG: it uses the standard definition of flow dependence (Definition 2.8) with object variables. The solution is to extend the definition of flow dependence to account for objects. Interestingly, this extension allows for novel situations. For instance, in Figure 3.5b, `a` in line 5 depends on `a` in line 3 (even though it is redefined in line 4), but also, a (partial) definition can depend on another definition. In particular, `a` in line 4, which is a (partial) definition, depends on `a` in line 3, which is another definition.

With all these ideas, we identify (and formally define) two new dependences that complement the classical flow dependence with a specific treatment for object variables (it does not apply to primitive variables, which continue using the classic flow dependence definition (Definition 2.8)). We call these new dependences *object-flow dependence* and *object-reference dependence*. The former can be formally defined with an extended CFG.

Definition 3.3 (Extended Control-Flow Graph). *Given a CFG $G = (N, A)$, its extended version eCFG is a graph G' with the same nodes and edges as G with one exception: every node $n \in N$ that defines an object variable and contains $m > 1$ variable definitions ($DEF_{os}(n) = [v_1, v_2 \dots v_m]$) is replaced by a sequence of nodes ($n_1, \dots, n_m$, connected by edges) where each variable definition is represented by one single node.*

The eCFG is useful to explicitly represent the order in which a sequence of operations happen inside a single statement. These operations cannot be distinguished in a CFG because all of them are represented with one single node, while they can be distinguished in the eCFG because they are represented by a sequence of nodes.

Example 3.5. *Consider the assignment* `A a1 = new A(1,2)` *in line 11 of Figure 4.1. This assignment contains a total definition of variable* `a1` *and its CFG node together with its sequence of definitions and uses are shown in Figure 3.6a. Note that the sequence of definitions appear in the order described by scenario 1 for total definitions. Since the node contains three different definitions, the node must be split into different nodes in the corresponding eCFG. Figure 3.6b shows how the initial CFG node is split into three different nodes (one per variable definition) which are sequentially connected to form the corresponding eCFG representation.*



(A) CFG node (B) eCFG nodes

FIGURE 3.6: CFG and eCFG nodes corresponding to statement `A a1 = new A(1,2)`

We can now formally define object-flow dependence.

Definition 3.4 (Object-Flow Dependence)**.** *Let m and n be nodes in an eCFG. n is* object-flow dependent *on m if:*

#1 (i) n defines an object o,

(ii) m uses the object o, and

(iii) there exists a control-flow path from n to m where object o is not redefined.

or

#2 (i) n defines a data member x of an object o,

(ii) m defines object o[3], and

(iii) there exists a control-flow path from n to m where data member x of o is not redefined.

[3]I.e., in the eCFG, $DEF_{os}(m) = [v]$, where v is an object variable that points to object o.

The first set of conditions corresponds to the classic definition of flow dependence, the use-definition dependence, which also applies to object variables, even if the definition is partial. The second one considers a definition-definition dependence, produced by partial definitions. This flow dependence considers the case when the slicing criterion is an object variable, and it depends on all the complementary partial definitions that, together, produce the complete value of that variable.

Object-flow dependence represents all situations in which the values of the data members of an object are propagated. In particular, it is able to identify multiple partial definitions of an object and properly connect all of them to produce the whole value of an object. But object-flow dependence does not consider the *object reference* of objects. Unlike primitive variables, object variables have a pointer to a memory position where a specific object is stored. This pointer is updated every time a total definition of an object variable is executed (see Definition 3.1). Example 3.6 shows a scenario where object-flow dependence is insufficient to include the reference of an object in a slice.

Example 3.6. *Consider the code in Figure 3.7, where an object* c1 *of class C (which has one single data member x) is totally defined in line 2. Then, line 3 redefines data member x (even though its unique data member is redefined, the object reference remains unchanged). Finally, line 4 contains a use of variable* c1*. All object-flow dependences in this code are represented in the graph of the figure labelled with #1 and #2. These object-flow dependences are the ones generated by the first (#1) and second (#2) sets of conditions in Definition 3.4, respectively.*

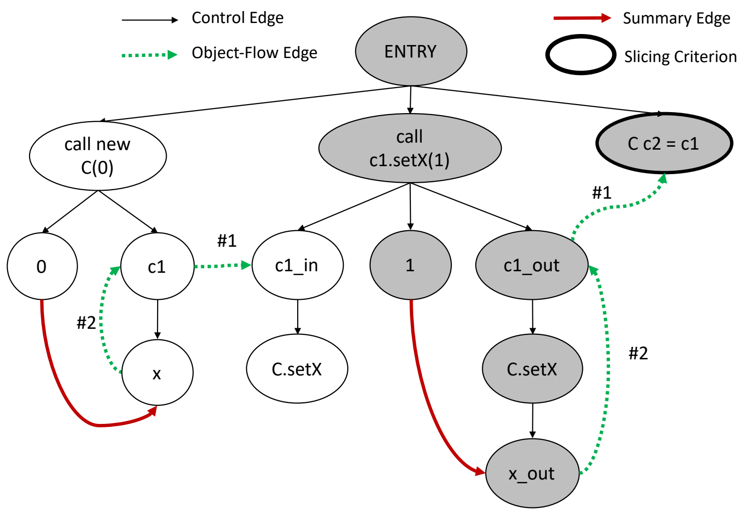
With these dependences, the slice computed for $\langle 4, \mathrm{c1}\rangle$ is shown with grey nodes. Clearly, this slice is incomplete, because line 2 should be included in the slice. The problem is that there is a missing dependence between c1_out *in line 3 and* c1 *in line 2. The missing part is the object reference of* c1*, which is defined in line 2. The rationale is the following: even though all data members of* c1_out *are defined in line 3, the object reference of* c1_out *is not. Therefore,* c1_out *in line 3 must depend on the code that defines its object reference (line 2).*

To account for these missing dependences, we define a new type of dependence called *object-reference dependence* that complements object-flow dependence. It connects objects with their object reference. Formally,

Definition 3.5 (Object-Reference Dependence)**.** *Let G be a CFG. Let m and n be nodes in G. n is* object-reference dependent *on m if m totally defines an object o, n partially defines object o, and there exists a control-flow path from m to n where object o is not totally redefined.*

Object-flow and object-reference dependences are the key elements to solve the incompleteness problem of the JSysDG. The completeness of both dependences with respect to any combination of DEF/USE statements is an important result that we formulate and prove hereunder.

Before proving the completeness of object-flow dependences, in order to ease the proof by reducing the number of possible scenarios, we enunciate and prove the following lemma:



```
(...)
C c1 = new C(0);
c1.setX(1);
C c2 = c1;
```

FIGURE 3.7: Code with data member redefinition (left) and its JSysDG with object-flow (right)

Lemma 3.1. *Let s be a statement in a program P. Let v be an object variable totally defined at s that points to object o. The value of v at s is not object-flow dependent on any previous statement s'.*

Proof. According to Definition 3.1 (see also the scenarios 1 and 2 of Section 3.2), a statement s that totally defines an object variable also defines all its data members. In Definition 3.4, a statement s can be object-flow dependent on another statement s' iff: (i) s uses an object o, or (ii) s defines an object o and there is a data member x of o not defined between s' and s. Since s is a definition of o and also defines all o's data members, neither (i) nor (ii) are fulfilled and, thus, s cannot be object-flow dependent on any previous statement. □

Now, we can formally enunciate and prove the theorem.

Theorem 3.1 (Object-Flow Completeness). *Let $s_1; s_2; \ldots s_n;$ be a sequence of statements. Let x be an object variable or a data member of an object variable defined at s_1. If the value of an object o at s_n data depends on the execution of s_1, then there is a transitive object-flow dependence between s_1 and s_n.*

Proof. First of all, according to Lemma 3.1, when s_n is a total definition the theorem is trivially proved because s_n defines the value of o and its data members and no object-flow path can end at s_n. With respect to the rest of possibilities, we divide the proof in three different scenarios:

1. **o is defined at s_1 and not redefined in $s_2 \ldots s_{n-1}$.**

   (a) If s_n uses object o and $x = o$, i.e., object variable x points to object o, the claim follows trivially by case #1 of Definition 3.4.

   (b) If s_n partially defines object o and x is a data member of o not defined at s_n the claim follows trivially by case #2 of Definition 3.4.

(c) If s_n uses object o and x is a data member of o, then s_1 also defines o according to scenario 3 associated to partial definitions in Section 3.2. Thus, the claim follows by the object-flow path formed by cases 1a and 1b.

(d) If s_n partially defines object o and $x = o$ there exists an object-flow path from s_1 to s_n iff s_1 also defines a data member of o not defined in s_n, which makes this case equivalent to case 1b.

2. **o is not defined at s_1.**

   If s_1 defines x where $x \neq o$ or x is a data member of a different object p, object-flow dependence cannot be applied, because in both cases #1 and #2 of Definition 3.4 it is mandatory for both statements to operate over the same object. Hence, there cannot be an object-flow path between s_1 and s_n.

3. **x is redefined in $s_2 \ldots s_{n-1}$.**

   In this case, we can assume that $x = o$ or x is a data member of object o because in any other case we will find ourselves in case 2. We can prove the claim for $n = 3$ ($s_1; s_2; s_3;$) because it does not matter the number of transitive dependences. The proof is the same for each transitive step. We can analyse all cases separately. We use the following notation: $s_i(A, B)$ with $A = D$ to denote that x is defined at s_i and with $A = U$ to denote that x is used at s_i; and with $B = O$ to denote that x is an object variable that points to object o at s_i and with $B = DM$ to denote that x is a data member of object o at s_i. Since s_1 and s_2 perform the same definition over x they must share the same notation in all cases. This fact leaves 8 possible scenarios:

   (a) $s_1(D, O); s_2(D, O); s_3(D, O);$

   (b) $s_1(D, DM); s_2(D, DM); s_3(D, O);$

   Scenarios 3a and 3b are trivially proved because s_3 is a total definition of object o and total definitions does not depend on any previous statement according to Lemma 3.1. Hence, s_3 cannot be object-flow dependent on a previous statement s_1.

   (c) $s_1(D, O); s_2(D, O); s_3(U, O);$

   (d) $s_1(D, O); s_2(D, O); s_3(D, DM);$

   In scenario 3c, s_3 is trivially object-flow dependent on s_2 (case #1 of Definition 3.4) while in scenario 3d s_3 is object-flow dependent on s_2 if s_2 defines a data member of o different from the one defined at s_3 (case #2 of Definition 3.4). In both scenarios there is not direct object-flow dependence between s_1 and s_3 because the redefinition of x in s_2 prevents it. Additionally, there cannot be a transitive dependence either because s_2 cannot be object-flow dependent on s_1 due to Lemma 3.1.

   (e) $s_1(D, DM); s_2(D, DM); s_3(U, O);$

(f) $s_1(D, DM); s_2(D, DM); s_3(D, DM);$

To proof scenarios 3e and 3f it is important to remember that when a statement s_i defines a data member of an object o, it also defines o (see scenario 3 associated to partial definitions in Section 3.2). In scenario 3e, s_3 is object-flow dependent on s_2 because it is the last existent definition of object o (case #1 of Definition 3.4). In scenario 3f, s_3 is object-flow dependent on s_2 if s_2 defines a data member of o different from the one defined at s_3 (case #2 of Definition 3.4). In scenarios 3e and 3f, due to the existence of s_2, s_3 cannot be directly dependent on s_1 according to Definition 3.4. In turn, in both scenarios s_2 is object-flow dependent on any previous statement that defines a different data member of o (case #2 of Definition 3.4). Since s_1 and s_2 define the same data member x s_2 is not object-flow dependent on s_1 and there is no transitive object-flow path between s_1 and s_3.

(g) $s_1(D, O); s_2(D, O); s_3(U, DM);$

(h) $s_1(D, DM); s_2(D, DM); s_3(U, DM);$

Considering scenarios 3g and 3h, note that the theorem considers "the value of an object o" at s_3. In these two cases, s_3 uses a data member of an object o. If the data member of o is itself an object, these scenarios would be equivalent to scenarios 3c and 3e respectively. Finally, if the data member o is a primitive, this case would be out of the scope of the proof. In this case, there would not be a path to the last definition of this data member formed by object-flow edges, but for flow edges.

□

On the other hand, we enunciate and prove Lemma 3.2, which will be used to ease the proof the completeness of object-reference dependences by reducing the number of possible scenarios.

Lemma 3.2. *Let s be a statement in a program P. Let v be an object variable totally defined at s that points to object o. The reference of v at s is not object-reference dependent on any previous statement s'.*

Proof. According to Definition 3.5, a statement s is object-reference dependent on another statement s' if s partially defines an object variable v. Since s totally defines v, s also defines v's reference (see Definition 3.1) and the condition is not fulfilled. Thus, s cannot be the target of any object-reference dependence. □

After proving Lemma 3.2, we can now formally define and prove the completeness of object-reference dependences.

Theorem 3.2 (Object-Reference Completeness). *Let $s_1; s_2; \ldots s_n;$ be a sequence of statements. Let x be an object variable totally defined at s_1. If the reference of an object variable v at s_n depends on the total definition of s_1, then there is a path formed by object-flow and/or object-reference edge between s_1 and s_n.*

Proof. First of all, according to Lemmas 3.1 and 3.2, when s_n is a total definition the theorem is trivially proved because s_n cannot be the target of any object-flow or object-reference dependence. With respect to the rest of possibilities, we divide the proof into three different scenarios:

1. **x is defined at s_1 and not redefined in $s_2 \ldots s_{n-1}$.**

   (a) If s_n uses object variable v and $x = v$ the claim follows trivially by the object-flow edge generated by case #1 of Definition 3.4.

   (b) If s_n partially defines object variable v and $x = v$ the claim follows trivially by Definition 3.5.

2. **v is not defined at s_1.**

   If s_1 defines x where $x \neq v$, object-flow and object-reference dependences cannot be applied, because in both Definitions 3.4 and 3.5 it is mandatory for both statements to operate over the same object. Hence, there cannot be a path formed by object-flow and/or object-reference edges between s_1 and s_n.

3. **x is redefined in $s_2 \ldots s_{n-1}$.**

   In this case, we can assume that $x = v$ because in any other case we will find ourselves in case 2. We can prove the claim for $n = 3$ $(s_1; s_2; s_3;)$ because it does not matter the number of transitive dependences. The proof is the same for each transitive step. We can analyse all cases separately. We use the following notation: $s_i(A)$ with $A = D_T$ to denote that x is totally defined at s_i, with $A = D_P$ to denote that x is partially defined at s_i, with $A = U$ to denote that x is used at s_i, and with $A = *$ to denote that A can be either D_T, D_P or U. There are several possible scenarios:

   (a) $s_1(*); s_2(*); s_3(D_T);$

   Scenario 3a represents the set of cases where s_3 totally defines variable v. These scenarios are trivially proved by Lemma 3.2, since s_3 cannot be object-referent dependent on any previous statement.

   (b) $s_1(D_P); s_2(*); s_3(*);$

   Scenario 3b illustrate the set of cases where s_1 partially defines x. Note that this scenario is not contemplated by the theorem because "x is totally defined at s_1" is a condition described in the theorem that this scenario does not contemplate.

   (c) $s_1(D_T); s_2(D_T); s_3(U);$

   (d) $s_1(D_T); s_2(D_T); s_3(D_P);$

   In scenarios 3c and 3d, s_3 is trivially object-flow dependent on s_2 by cases #1 and #2 of Definition 3.4, respectively. In both scenarios there is neither direct object-flow, nor object-reference dependence between s_1

and s_3 because the total definition of x in s_2 prevents it. Additionally, there cannot be a transitive dependence because s_2 cannot be object-flow nor object-reference dependent on s_1 due to Lemmas 3.1 and 3.2.

(e) $s_1(D_T); s_2(D_P); s_3(U);$

In scenario 3e, s_3 is trivially object-flow dependent on s_2 according to case #1 of Definition 3.4. Additionally, since s_2 partially defines an object variable that is totally defined in s_1, s_2 is object-reference dependent on s_1 according to Definition 3.5. Therefore, in this scenario, there is a transitive path formed by object-flow and object-reference edges that connect the use of an object variable to its last total definition in s_1.

(f) $s_1(D_T); s_2(D_P); s_3(D_P);$

Finally, in scenario 3f, s_3 may be object-flow dependent on s_2 according to case #2 of Definition 3.4 if they define different data members of the same object. Either way, both s_3 and s_2 are always object-reference dependent on s_1 for being partial definitions of an object variable totally defined at s_1. Consequently, there is at least one path from the last total definition in s_1 to the later partial definition in s_3 formed by an object-reference edge.

$\square$

The final JSysDG that includes our new dependences is shown in the following example. With object-reference dependence it solves the problem explained in Example 3.6.

Example 3.7. *Consider again the code in Figure 3.1a. Figure 3.8 shows its associated JSysDG. The upper part of the figure corresponds to the representation of class `Main`, where we can see how object dependences are also defined over the tree structure of method calls. Although object dependences add edges between object variables, the original definition of flow dependence is still applied to primitive variables. This happens in the call `a1.f(10)`, where data member `y` of the constructor call is linked to the argument-in node `y_in`. We can see that the object-reference edge is needed to connect the object variable `a1` after the call `a1.setX(3)` with the construction of this object in `call new A(1,2)`. This JSysDG extended with object dependences can properly slice the graph from any slicing criterion. For instance, the slice computed for the slicing criterion ⟨13, `a`⟩ in Figure 3.1a would be the expected slice, i.e., the code in black in Figure 3.2b.*

Slicing the graph with Object-Flow Dependence

The JSysDG slicing algorithm is the standard one proposed by Horwitz et al. [85]. It computes slices in two phases: (i) traverse the graph backwards from the slicing criterion collecting all nodes reached using any edge except *output* edges, (ii) traverse the graph backwards from any node in the slice collecting all nodes

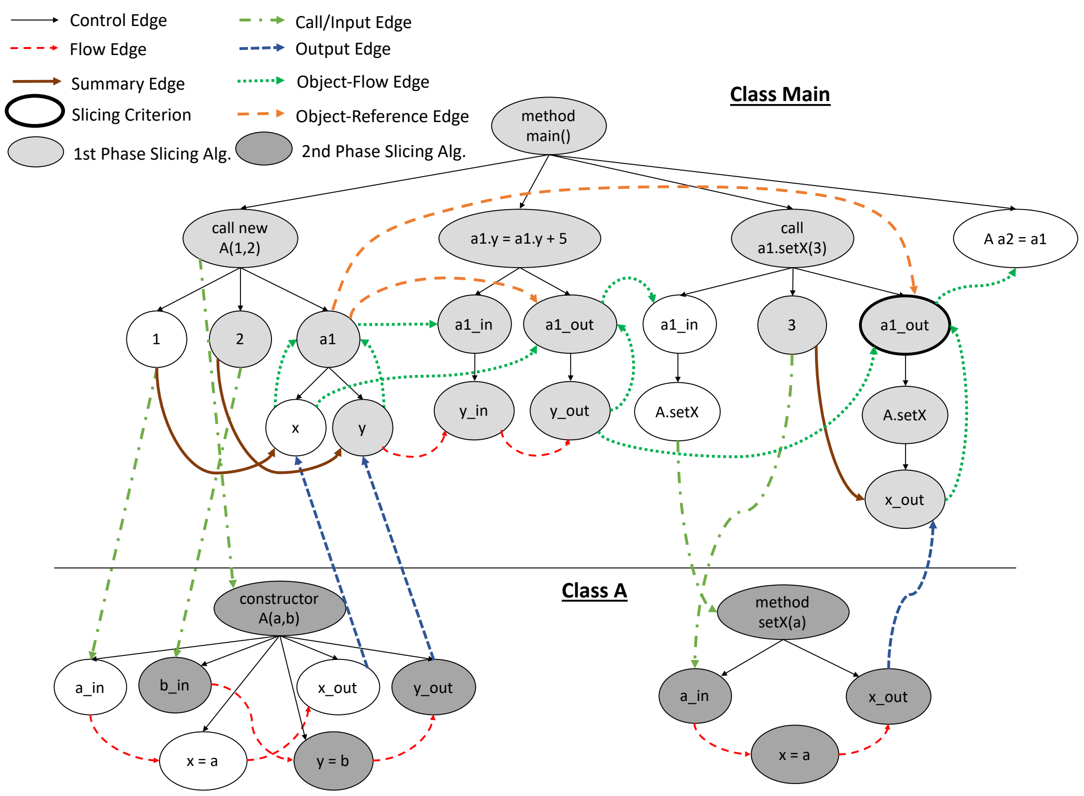


FIGURE 3.8: JSysDG of the program in Figure 4.1 and slice w.r.t. $\langle 13, \mathtt{a}\rangle$

reached using any edge except *call* and *input* edges. The overall process has a linear time complexity. This algorithm can be used with our graph, producing the same precision as with the JSysDG. However, this algorithm does not take advantage of the new object-flow dependences, producing a loss of precision. The standard algorithm would include in the slice all the data members of an object variable even if we are only interested in one of them. For instance, in Figure 3.8, if we consider the node y inside the method call *new* $A(1, 2)$, the algorithm would unnecessarily include in the slice data member x and its value 1 (due to edge $a1 \xleftarrow{\#2} x$).

The problem can be solved by limiting the traversal of the object-flow edges in certain cases. When the traversal reaches a node n, an incoming object-flow edge can only be traversed if one of the following three conditions is true:

1. n has been reached via an object-flow edge

2. n is the slicing criterion

3. n is a predicate

Condition 1 ensures that the traversal of object-flow edges is still transitive. Conditions 2 and 3 provide a starting point to traverse object-flow edges. When the slicing criterion is an object variable (Condition 2), we need to traverse all object-flow edges to include in the slice the value of all its data members. On the other hand, in Condition 3, when the traversal reaches a predicate (e.g., the condition of an `if` or a `while` statement) we need to follow object-flow edges to include the data members used by its condition in order to keep the slice complete. The restriction imposed to the traversal of object-flow edges increases the precision of the algorithm in the presence of object variables, while keeping its linear time complexity. Algorithm 3.1 includes all these conditions in the original slicing algorithm proposed in [85], making it suitable to traverse the JSysDG after adding object-flow dependences.

Algorithm 3.1 Slicing Algorithm for the JSysDG with Object-Flow Dependence

Input: A JSysDG 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: $S_0 \leftarrow \{\langle n_{sc}, none \rangle\}$.

```
 1: function MarkNodesOfSlice($G, n_{sc}$)
 2:   $S_1 \leftarrow$ MarkReachingNodes($G, S_0, n_{sc}, \{Output\}$)        ▷ Phase 1
 3:   $S_2 \leftarrow$ MarkReachingNodes($G, S_1, n_{sc}, \{Call, Input\}$)   ▷ Phase 2
 4:   $S \leftarrow \{n \mid \langle n, edgeType \rangle \in S_2\}$
 5:   return $S$

 6: function MarkReachingNodes($G, N, n_{sc}, EdgeTypes$)   ▷ Change ($A$)
 7:   $WorkList \leftarrow N$
 8:   while $WorkList \neq \emptyset$ do
 9:     select some $\langle n, lastEdgeType \rangle \in WorkList$   ▷ Change ($B$)
10:     $WorkList \leftarrow WorkList \setminus \langle n, lastEdgeType \rangle$
11:     for all $edge \in$ getIncomingEdges($n$) do
12:       if $edgeType \in EdgeTypes$ then
13:         continue
14:       $m \leftarrow$ getSourceNode($edge$)
15:       if $edgeType = ObjectFlow$ then
16:         if $lastEdgeType \neq ObjectFlow \wedge n \neq n_{sc} \wedge \neg$isPredicate($m$) then
17:           continue
18:       $N \leftarrow N \cup \langle m, edgeType \rangle$
19:       $WorkList \leftarrow WorkList \cup \langle m, edgeType \rangle$
20:   return $N$
```

Algorithm 3.1 illustrates the slicing process by including a couple of relevant changes to Horwitz's algorithm. The first one (A) is that the slicing criterion n_{sc} is now used as a parameter of function MarkReachingNodes, since this function needs to identify in Line 16 whether the current node is the slicing criterion (to check Condition 2). Additionally, (B) the `WorkList` in function MarkReachingNodes contains now tuples of two elements (see Line 9) instead of a single element, storing also the information about the type of the last traversed edge (to check Condition 1). The three aforementioned conditions are checked in lines 15-17. If any of them holds then the current node is not included in the slice.

Example 3.8 shows how the application of Algorithm 3.1 solves the incompleteness problem of the JSysDG presented in Section 3.1.

Example 3.8. *The graph in Figure 3.8 represents the JSysDG of the code in Figure 3.1a augmented with object-flow and object-reference dependences. It shows the slice computed with respect to the object variable* `a1` *in line 13 after the method call* `a1.setX()` *(the slicing criterion node is marked with a bold line in the graph). To clearly show the two traversal phases of the algorithm, the slice is divided into two sets of nodes. The nodes marked in light grey are added to the slice during Phase 1. On the other hand, the dark grey nodes are the nodes added to the slice during Phase 2. The slice code is exactly the expected slice shown in Figure 3.2b. It is worth mentioning that the change proposed to Horwitz et al.'s algorithm has a direct impact on the slice's precision.*

Note that the traversal algorithm improves its accuracy by preventing the `x` *data member of object variable* `a1` *in method call* `new A(1,2)` *to be included in the slice. This happens because* `x` *can only be reached from* `a1` *(through an object-flow edge) and this edge can only be traversed from another object-flow edge, according to Condition 1 of the slicing algorithm. Even though* `a1` *can be reached from two object-flow edges, none of them belong to the slice, thus* `x` *is never included in the slice.*

3.3 Implementation

The proposals described in this chapter have been implemented in a program slicer called JavaSlicer. JavaSlicer is a program slicer for Java implemented in Java that uses the JSysDG as the base graph. Additionally, the slicer includes the representation of object-oriented programs described in Chapter 3, which implies the representation of object-flow and object-reference dependences. The slicer allows us to select object variables in an OO program as the slicing criterion, obtaining an accurate slice as a result, which includes the value of all the object data members and also the reference of the object variable itself. The JavaSlicer project is composed of 2 different main modules: *sdg-cli* and *sdg-core*. The sdg-cli module contains the presentation layer of the tool and is responsible of the reception and management of the input arguments given by the user, as well as providing the user with the resulting slice. On the other hand, sdg-core represents the logical layer, dealing with the whole graph representation and generating the resulting slice. The project is divided in 101 Java classes, distributed into 21 different packages, which include more than 7600 lines of code. The source code is publicly available at `https://github.com/mistupv/JavaSlicer` and a limited online version to test the tool can be found at `https://mist.dsic.upv.es/JavaSlicer/demo`.

Figure 3.9 illustrates an overall idea about the communication between the modules and classes in JavaSlicer. The figure shows the most relevant classes of both slicer modules and how they communicate in order to compute a slice from a Java program. In the figure, we differentiate two main modules in the implementation (represented with rounded squares): the module that implements the presentation layer of the slicer (sdg-cli) and the module that specifies the logical layer of the slicer (sdg-core). Additionally, in Figure 3.9, solid squares represent Java classes inside these modules, solid arrows represent calls between classes (sometimes between different modules), and dashed arrows represent the input/output resources of the slicer. Finally, dashed squares represent external Java modules called during the slicing process.

The implementation contains more than 100 classes but, to keep the representation readable and simple, Figure 3.9 only shows the communication between the 12 most relevant classes to make an overall description of the whole process. Each one of the represented classes and their main functionality is the following:

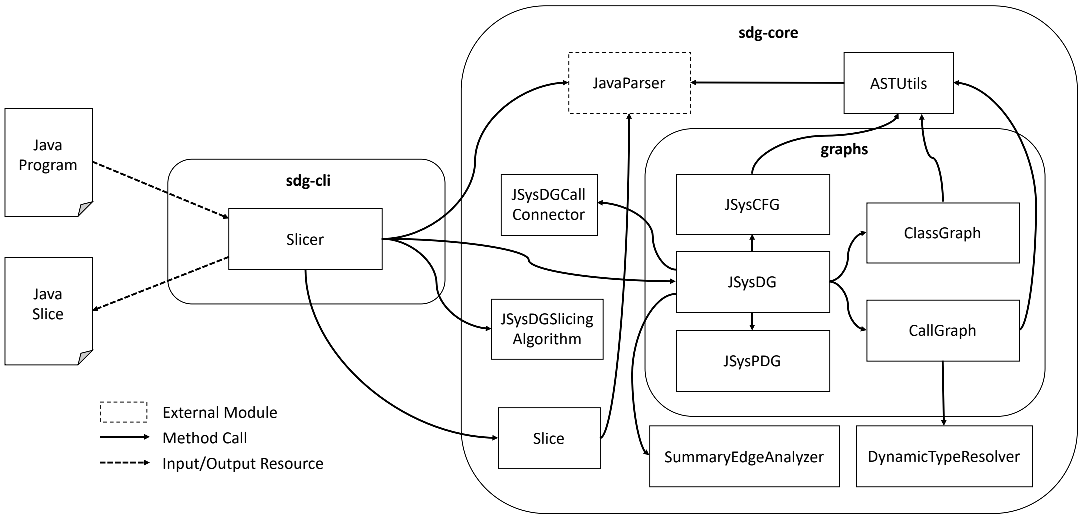


FIGURE 3.9: JavaSlicer architecture and communication between modules and classes

- `Slicer`. This class orchestrates the whole slicing process. It receives the input given by the user, configures and runs JavaParser to extract the AST of the input Java project, build its JSysDG, runs the slicing algorithm over the generated graph with the selected slicing criterion, and finally transforms the obtained slice back to code. In this class, the resources generated by each subprocess are given to the next one in order to compute the final slice.

- `ASTUtils`. This class contains a collection of functions that make the communication between the rest of classes and JavaParser easier. These functions extract a set of features and relationships that are contained inside the JavaParser complex structure.

- `JSysDG`. This class manages the construction process of the JSysDG. The class receives the JavaParser information of all classes and processes all their elements. It is worth to mention that, although different graphs are built during the process, the nodes that represent each statement of the code are shared by them, i.e., the nodes that represent program statement are only generated once. The construction of the JSysDG is performed in 6 steps:

1. The class graph of the whole project is computed.
2. The CFGs of every method in the code are built.
3. The call graph is computed.
4. Each PDG is computed by using their previously generated CFG. These computation includes all the new proposed dependences: object-flow and object-reference.
5. Interprocedural edges are generated by connecting method calls to all their potentially called (polymorphism) method declarations.
6. Summary edges are computed for the result and every redefined argument inside a method declaration.

- `ClassGraph`. The `ClassGraph` class builds and represents the class graph of the project. Each class node inside this graph contains a set of children nodes that represent data members or method declarations inside the class. Additionally, the class graph represents the pair of classes related by any type of inheritance relationship.

- `JSysCFG`. This class traverses the whole structure of the AST provided by JavaParser for all methods. It builds a CFG for every program method, which contains a CFG node per statement. Additionally, it adds the definition and use sets of variables in each node and creates control flow edges between the generated statements, representing all the possible execution paths.

- `CallGraph`. This class builds and represents the call graph of the analysed Java project. In this graph, each node representing a method call is linked with a call edge to all the potentially called definitions. Polymorphism is a key factor in this graph. All the possible dynamic types of the scope of each method call need to be computed to perform a complete static representation of the program. This dynamic type computation is performed by the `DynamicTypeResolver` class.

- `DynamicTypeResolver`. This class extracts, by traversing the class graph, the set of possible dynamic types of an object variable in a particular program statement.

- `JSysPDG`. This class, builds the PDG of each method declaration by using its associated CFG. To this aim, it applies the corresponding algorithms to compute the control, flow, object-flow, and object-reference dependences. Additionally, this class also analyses method calls to extract the associated definitions and uses contained in each method declaration. Finally, this class unfolds those objects in scopes and arguments into their corresponding tree representations, leaving the PDG ready for interprocedural connection.

- `JSysDGCallConnector`. This class uses the call graph to interprocedurally connect all the arguments in method calls with their corresponding parameters in method declarations. This connection is also extended to those nodes that represent objects, connecting the corresponding trees in both method calls and method declarations.

- `SummaryEdgeAnalyzer`. The `SummaryEdgeAnalyzer` class applies Algorithm 3.1 intraprocedurally in every method, computing the formal-in dependences for each formal-out of the method, and generating the so-called summary edges.

- `JsysDGSlicingAlgorithm`. This class implements Algorithm 3.1 and runs it for a given slicing criterion, obtaining as a result the set of JSysDG nodes that conform the program slice.

- `Slice`. This class contains the set of nodes that represent the slice, and is the class responsible for transforming these set of nodes into the corresponding Java code by successive calls to the functions in JavaParser libraries.

These 12 classes are just the top of the iceberg, where the main processes to compute the program slice are performed. The other 89 classes of the project are used to fulfil necessary implementation processes like: the process to extract argument-in-parameter-in assignments into new nodes when building PDGs, all the classes used to generate the inner structure of the graph with all the different types of nodes and edges, or all the classes used to visit the whole JavaParser structure needed to build the CFG. The project source code together with a set of example programs can be found in the public git repository of the JavaSlicer tool:

`https://github.com/mistupv/JavaSlicer`

3.4 Experimental Results

In this section we compare our implementation, which include object-flow and object-reference dependences, with the original JSysDG. We have compared both graph implementations by measuring the graph generation time, the slicing time, and the size of the slice, comparing the results obtained by both slicer executions.

All the algorithms and ideas described in this paper have been implemented in a prototype slicer for Java programs called *JavaSlicer*, and has been released as libre software[4].

To compare the performance difference between the JSysDG and JavaSlicer, we used the publicly available Java library *re2j*, a library developed by Google to work with regular expressions in Java. In particular, we used the most recent release of this library (version 1.6[5]). *re2j* is a project with 8100 lines of code, distributed in 19 different Java files. In order to evaluate the techniques proposed throughout this work, we used both the JSysDG and JavaSlicer to build and slice the whole *re2j* library. Then, to make the comparison meaningful, we

[4]Available at `https://github.com/mistupv/JavaSlicer`

[5]Available at `https://github.com/google/re2j/releases/tag/re2j-1.6`

Size Range	SCs	Slice Time (A)	Slice Time (B)	Slower	Size (A)	Size (B)	Improv.
[0, 100)	49	$0.076 \pm 0.001ms$	$0.927 \pm 0.018ms$	48.876	9.10	39.59	693.74%
[100, 1000)	95	$130.98 \pm 1.823ms$	$217.315 \pm 2.874ms$	0.782	608.69	738.68	23.19%
[1000, 1400)	122	$622.67 \pm 10.833ms$	$1164.423 \pm 13.093ms$	0.857	1284.66	1664.99	29.32%
[1400, 1800)	146	$881.09 \pm 13.101ms$	$1584.023 \pm 12.429ms$	0.779	1535.62	1948.16	26.77%
[1800, ∞)	31	$1603.05 \pm 11.769ms$	$2943.965 \pm 15.702ms$	0.842	1994.42	2522.74	26.73%
[0, ∞)	443	$602.13 \pm 9.078ms$	$1095.440 \pm 12.039ms$	6.126	1130.99	1439.91	100.47%

Build times	JSysDG: $10.85 \pm 0.09s$	JavaSlicer: $13.35 \pm 0.07s$ (23% slower)

TABLE 3.2: Summary of experimental results, comparing the slices of *re2j* produced by the JSysDG (A) and JavaSlicer (B).

selected as slicing criteria all the object variables (root node of the object tree representations) that are directly connected to a return statement[6]. The result of this selection is a total of 443 different slicing criteria located over the 19 Java files.

All experiments were done on an Intel Core i5-7600 processor with 16 GB RAM (DDR4 at 2400MT/s) under a Linux OS with kernel version 5.18.1. The experiment was run with the OpenJDK Java Virtual Machine (version 11.0.15+10). During the execution of the experiment, all processes and services except for the program slicer and the shell it was launched from were completely stopped to prevent interferences in the CPU performance.

The methodology used to measure the performance of both slicers was the following: each time measurement (the graph generation or a specific slice) was repeated 101 times, and the first iteration was always discarded (to avoid influence of dynamically loading libraries to physical memory, data persisting in the disk cache, etc.). Finally, we computed the average with error margins (with 99% confidence) with the remaining 100 values.

The results of the experiments performed are summarised in Table 3.2[7]. In it, the slicing criteria are grouped according to the size of the JSysDG slice (in nodes). This is due to the strong influence the size of the slice has on the time required to compute it and on the relative sizes of slices produced by both graphs. The columns of Table 3.2 are described as follows:

- **Size Range**: the sizes of the JSysDG slices (in nodes) that are contained in each row.
- **SCs**: the number of slicing criteria contained in a particular range.
- **Slice Time (A/B)**: the average time required to slice the corresponding JSysDG (A) and JavaSlicer (B).
- **Slower**: how much slower is JavaSlicer in comparison to the JSysDG. It is computed as $(Time_B - Time_A)/Time_A$.

[6]Note that, to compare the performance of both models, selecting an object variable during the slicing traversal is preferable. Any slicing criterion that does not include object variables would produce two identical slices from both graphs, as the two representations are identical except for object-flow and object-reference.

[7]The full dataset produced by the slicer execution is publicly available at `https://mist.dsic.upv.es/git/program-slicing/SDG/-/snippets/9`.

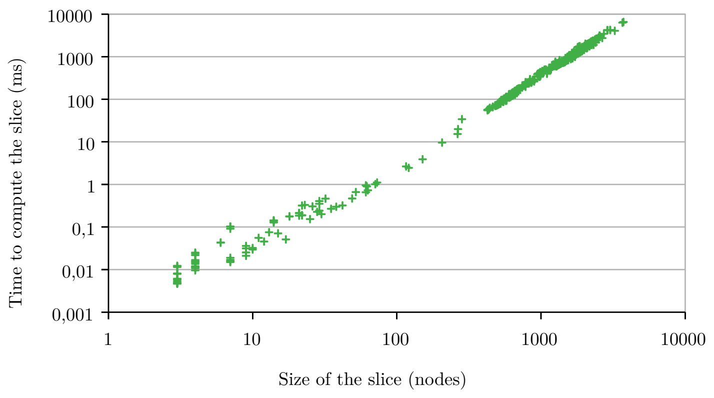


FIGURE 3.10: Relationship between the size and time required to compute a slice (logarithmic scale)

- **Size (A/B)**: the average number of nodes in the JSysDG (A) and JavaSlicer (B) slices.
- **Improv.** (*improvement*): the average increase in size of the JavaSlicer slice compared to the JSysDG slice. It is computed as $(Size_B - Size_A)/Size_A$.

To interpret the results, we first need to focus on the usage of the SDG. Typically, a graph will be built once and sliced multiple times, and thus its creation can be (and often is) much more costly than its traversal. Regarding complexity, creation is polynomial and traversal is linear. Our results are as expected: creating the graph is between one and four orders of magnitude more time-consuming (depending on the final slice's size). Regarding the graph creation, we can observe a 23% increase between the JSysDG and JavaSlicer, which is attributable to the great number of object trees that are featured on the graph. The graph itself consists of 42000 nodes, most of which represent objects, their polymorphism or their members. Thus, computing object-flow and object-reference edges represents a noticeable increase in time consumption.

If we turn our attention to the slicing portion of the results, we can see that the slices can be classified into two distinct groups. The first group consists of slices whose size is below 100 in the JSysDG tend to "blow up", as the addition of object-flow and object-reference adds a significant number of nodes to the slice (relative to the size of the original slice). Thus, the relative columns of the first row of results show a slicer that includes almost 700% more nodes and takes about 50 times longer. Due to the small size of the resulting slices and the short time it takes the JSysDG to compute them ($0.08ms$), a 50x increase only manages to bring the average up to $0.9ms$, which is negligible in most applications of program slicing. This first group can be considered an outlier, and it affects the averages of the table as a whole. The second group is represented by the rest of the table (slices with sizes above 100 nodes), where the results show that slices increase a 26.69% in average, with the cost being a 80% time increase. Throughout the table, time has increased linearly with slice size, as can be seen in Figure 3.10. Whether the cost is worth the improvement in completeness is up to the application of slicing. For example, in compiler optimisation, completeness is an important requirement for slicers; while for dependency highlighting or light debugging, a faster but incomplete slice may be more appropriate. Another important remark is that the slices produced by JavaSlicer are always larger or equal to those generated by the JSysDG, because the only difference between them is the addition of new dependences in the JavaSlicer.

3.5 Related Work

In 1996, Larsen and Harrold [112] proposed the first graph representation able to slice OO programs, the Class Dependence Graph (ClDG). Their approach was the first to provide a way to represent inheritance, polymorphism, and dynamic binding. The ClDG connected all the methods and data members of a class in a single graph. Their proposal was later improved by other approaches like those of Tonella et al. [196], or Liang and Harrold [118]. Although all these proposals focused on OO programs, none of them noticed the difference between total and partial definitions in object variables and their impact in flow/data dependence.

When we review the literature looking for specific slicing approaches for Java, we find interesting related papers. For instance, Hammer and Snelting [75] defined a new object unfolding process for method calls in presence of recursive data types, improving the results of the *k-limiting* approach proposed in [118]. They defined an algorithm to completely unfold object variables without unfolding the same object twice in the same object tree and without loosing dependences based on point-to information. Kashima et al. [96] compared four different backward slicing techniques for Java: static execute before (SBE), based in the CFG; context-insensitive slicing (CIS), ignoring the call context; hybrid model (HYB), where the slice is defined as the intersection of SBE and CIS; and improved slicing (IMP), based on the Hammer and Snelting's work [75]. They compared their precision, scalability, and tradeoffs, determining IMP as the more accurate but not applicable to large programs.

Other techniques focused on the representation of polymorphic objects in Java. This is the case of the JSysDG [202] and the SSLDG [128], mentioned at the beginning of this paper. Both graphs include a multiple-layer representation that contemplates Java programs at different levels: package level, class level, method level, and statement level. At statement level, both graphs use the unfolding of object variables in method calls using a tree-like representation for data members proposed in [118]. There are three main differences between these two graphs: the first one in that the SSLDG includes a sub-statement layer where object variables outside method calls are further split into their data members, making the representation of direct accesses to object fields explicit in the graph; the second one is that the SSLDG proposal enhances the information of the slicing criterion when slicing polymorphic objects, using a triplet as slicing criterion where the new element indicates the dynamic type of the object being sliced to further reduce the computed slice; and the third difference is the modification of the slicing algorithm to adapt its graph for forward slicing.

Other works dealt with object-oriented programs in other programming languages like C++ or Python. In [150], Pani et al. present an algorithm for finding dynamic slices for object oriented programs in presence of function overloading on C++; and in [91], Jain and Poonia proposed a mixed static and dynamic slicing for C++ OO programs, where they generate dynamic slices in a faster and more accurate way by using object-oriented information in C++. These works are centred on dynamic slicing, and they are not focused in the representation of object variables, but in taking advantage of the information given by the compiler to compute the slice. On the other hand, a program slicing approach for Python is presented by Xu et al. in [209]. This work defines the Python System Dependence Graph (PySDG), used to slice Python First-Class objects. The PySDG takes all the first-class objects including functions, methods, classes and modules into consideration, to construct the dependencies between the definition and use statements of these first-class objects. In this model, the authors also introduce a new kind of dependence (*Entity Dependence*), which describes the dependency relationship between the statement which defines the entity object and the statement which calls the entity object.

There are also some other works mainly focused on modelling data dependences for slicing in object-oriented programs. Chen and Xu [36] augmented the PDG of each method with tags, used to distinguish the different definitions and dependences inside a statement. The authors defined five different sets: $\mathit{Def}(s)$, $\mathit{Ref}(s)$, $\mathit{Def}(s,x)$, $\mathit{Dep\_D}(s,x)$, and $\mathit{Dep\_R}(s,x)$. These sets were used to annotate data dependence edges with the program variables involved in each data dependence. Despite the perspective is interesting, its purpose is different to ours. It gives extra information to data dependences by annotating them, limiting the graph traversal at slicing level when reaching a node looking for just a particular variable. Contrarily, our approach focuses on establishing a dependence between an object variable and the value of all its corresponding data members in any program point. The work by Orso et al. [146] exhaustively analyses data dependence in the presence of pointers. They considered two different aspects: the classification of definitions and uses, and the classification of different kinds of paths in the CFG. In their work, they differentiate 24 kinds of data dependence and allowed the possibility of slicing with respect to only some of them including the considered dependences as part of the slicing criterion. Unfortunately, their data dependence is more suitable for point-to analysis than for the OO paradigm, as it is based on point-to relationships.

Our approach may seem similar to object slicing, introduced by Liang and Harrold [118], or class slicing, defined by Chen and Xu in [37], but there are some differences between their approaches and ours. In object slicing, the slicing criterion is defined with a tuple $\langle v, p, o\rangle$ where v is a variable in a program statement p, and o is an object variable of the program. Object slicing determines which statements of o's class affect the slicing criterion through object variable o. First, a standard slice is computed for the criterion $\langle v, p\rangle$. Then, considering the computed slice, a new process isolates all method calls with o as the caller object. Afterwards, method definitions corresponding to o's detected method calls are identified. Finally, o's slice is computed by extracting from the initial slice all the statements corresponding to those method definitions. Note that, in object slicing, the slicing criterion is not the object variable itself, but a mechanism to reduce the statements included in the original slice. Our objective is different: we are interested in considering the object variable o as the slicing criterion, extracting from the whole program (not only from o's class) the code that affects its whole value (the value of all its data members) in a particular statement. Class slicing [37] is similar to object slicing: it is also defined over a slicing criterion $\langle v, p, c\rangle$, but this time, instead of a specific object o, a class c is selected. Class slicing is restricted to a single class. It extracts any statement in c that affects the slicing criterion $\langle v, p\rangle$ for every object instance of class c that is part of the slice. This approach is a bit far from what we are interested in, because it considers a set of objects, not a single one and, once again, it focuses on a single class while we are interested in the whole program.

Chapter 4

Field-Sensitive Slicing with Constrained Graphs

Although the PDG has been extended several times to represent features like arbitrary control-flow [13, 65, 109]; exception handling [6, 64, 92, 93]; inter-procedural behaviour [16, 17, 41, 110]; or concurrency [36, 104]; among others, there is still a largely unaddressed problem that is a source of imprecision and that affects all programming languages: the slicing of composite data structures.

In this chapter, we show that the granularity level of the program dependence graph (PDG) to deal with composite data structures (tuples, lists, structures, objects, etc.) is inaccurate when slicing their inner elements, and we present the Constrained-Edges PDG (CE-PDG) that addresses this accuracy problem. The CE-PDG enhances the representation of composite data structures by decomposing statements into a set of nodes that represent the inner elements of the structure. Additionally, the CE-PDG introduces a second extension, the inclusion and propagation of data constraints through the CE-PDG edges, which allows for precisely slicing complex data structures. Both extensions are conservative with respect to the PDG, in the sense that all slicing criteria that can be specified in the PDG can be also specified in the CE-PDG, and the slices produced with the CE-PDG are always smaller or equal to the slices produced by the PDG.

Consider the four fragments of code with different data structures shown in Figure 4.1. We are interested in the values computed at the slicing criterion (the underlined variable in blue). The only part of the code that can affect the slicing criterion (i.e., the minimal slice) is the part of the code in black. Nevertheless, the slice computed with the PDG includes also the part in grey, which should be sliced, in the four cases.

In some cases, it is possible to solve the situation with a program transformation [18, 99, 100]. For instance, in Figures 4.1a and 4.1c we could apply these transformations:

```
person = {"John",36};  →  person.name = "John";
                          person.age = 36;

  nums = {2,arg,27};   →  nums[0] = 2;
                          nums[1] = arg;
                          nums[2] = 27;
```

```
foo(){
  struct data
  {
    string name;
    int age;
  };
  data person = {"John",36};
  int maxAge = person.age;
  printf("\%i",maxAge});
}
```

(A) Records (C++)

```
enum Light {
  Red    = 0,
  Yellow = 1,
  Green  = 2
}
void Main(){
  Light pass = Light.Yellow
             | Light.Green;
  Console.WriteLine(pass);
}
```

(B) Enums (C#)

```
void foo(int arg){
  int[] nums = {2,arg,27};
  int x = nums[2];
  System.out.println(x);
}
```

(C) Arrays (Java)

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
print(p1.age)
```

(D) Objects (Python)

FIGURE 4.1: Slicing composite data structures (slicing criterion underlined and blue, minimal slice in black)

With these transformations, we can decompose each data structure into its components avoiding the problem by using the qualified name `person.age` or the indexed array `nums[0]` as the name of an independent variable [18]. This transformation is called *atomization* [163]. An alternative approach uses the AST nodes as PDG nodes [180]. Unfortunately, despite solving several problems, these approaches fail to resolve the most important problems such as recursive (infinite) data types and the problem of pattern matching. For this reason, we have selected Erlang as the target language: a programming language that presents these two features. Erlang is a functional language that implements pattern matching and often presents a heavy use of tuples and lists, a (potentially infinite) recursive data type.

```
1  foo(X,Y) ->
2    {A,B} = {X,Y},
3    Z = {[7],A},
4    {[C],D} = Z.
```

(A) Original Program and PDG Slice

```
1  foo(X,Y) ->
2    {A,B} = {X,Y},
3    Z = {[7],A},
4    {[C],D} = Z.
```

(B) Minimal Slice

FIGURE 4.2: Slicing Erlang tuples (slicing criterion underlined and blue, slice in black)

Example 4.1. *Consider the fragment of Erlang code in Figure 4.2a, where we are interested in the values computed at variable* `C` *(the slicing criterion is* $\langle 4, \texttt{C} \rangle$*). The only part of the code that can affect the values at* `C` *(i.e., the minimal slice) is coloured in black in Figure 4.2b. Nevertheless, the slice computed with* *the PDG (shown in Figure 4.2a) contains the whole program. This is again a potential source of more imprecisions outside this function because it wrongly includes in the slice the parameters of function `foo` and, thus, in calls to `foo` their arguments and the code in which they depend are also included.*

The fundamental problem in this particular example is pattern matching: a whole data structure (the tuple `{[7],A}`) has been collapsed to a variable (`Z`) and then expanded again (`{[C],D}`). The traditional PDG represents that `{C}` flow depends on `Z`, and in turn, `Z` flow depends on `A`. Because flow dependence is transitive, slicing the PDG wrongly infers that `C` depends on `A`. This problem becomes worse in presence of recursive data types. For instance, trees or objects (consider a class `A` with a field of type `A`, which produces an infinite data type) can prevent the slicer to know statically what part of the collapsed structure is needed. An interesting discussion and example about this problem can be found in [186, pp. 2–3].

We propose a general method that solves the problem of accurately representing and slicing composite data structures. The method is applicable to any composite data structure (it produces the minimal slice for all the programs in Figures 4.1 and 4.2). The key ideas are (i) to expand the PDG with new nodes to precisely represent the subexpressions of the data structures, and (ii) introducing the concept of *constrained edges*: we label the PDG edges with structural information about the data structures, so that this information is used at slicing time to know exactly what edges should be traversed. We call the new resulting graph the CE-PDG (Section 4.1). Then, (iii) we provide a new slicing algorithm that takes advantage of the labels in the edges, limiting the traversal when necessary (because flow dependence is not transitive when applied to complex structures), and obtaining more accurate slices in the presence of composite data structures (Section 4.3). After that, (iv) we show how both model and algorithm can be adapted to the interprocedural part, defining the Constrained-Edges SDG (CE-SDG), and accurately slicing complex structures passed between methods through function calls (Section 4.4). Finally, we present an implementation of the model together with an experimental evaluation in Sections 4.5 and 4.6.

4.1 The CE-PDG

The key idea of the CE-PDG is to expand all those PDG nodes where a composite data structure is collapsed or expanded. This expansion augments the PDG with a tree representation for composite data structures, similar to the structure used to represent object parameters in OO languages (see Section 2.3 in Chapter 2). We describe how this structure is generated and we introduce a new kind of dependence edge used to build this tree structure. For this, we formally define the concepts of constraint and constrained edge; describing the different types, and how they affect the graph traversal during the slicing process.

Figure 4.2a shows that PDGs are not accurate enough to differentiate the elements of composite structures. For instance, the whole statement in line 4 is represented by a single node, so it is not possible to distinguish the data structure `{A,B}` nor its internal subexpressions. This can be solved by transforming the PDG into a CE-PDG.

The CE-PDG is not a graph constructed from scratch, but a specialisation of the PDG, i.e., it uses a PDG as the base graph and augments its expressivity to accurately treat composite structures. The transformation from a PDG to a CE-PDG is made following three sequential steps.

1. **Composite data decomposition**. The first step is the decomposition of all nodes that contain composite data structures so that each component is represented by an independent node. As in most Abstract Syntax Trees (ASTs), we represent data structures with a tree-like representation (similar to the one used in object-oriented programs to represent objects in calls [118, 202]).

   This process is recursive because it unfolds the composite structure by levels, i.e., if a subelement is another composite structure, it is recursively unfolded until the whole syntax structure is represented in the tree. In contrast to the PDG nodes (which represent complete statements), the nodes of this tree structure represent expressions. Therefore, we need a new kind of edge to connect these intra-statement nodes. We call these edges *structural edges* because they represent the syntactical structure.

   Definition 4.1 (Structural Edge). *Given a CE-PDG $P = \{N, E\}$, and two CE-PDG nodes $n, n' \in N$, there exists a* structural edge $n \dashrightarrow n'$ *if and only if:*

   - *n contains a data structure for which n' is a subexpression.*
   - *$\forall n, n', n'' \in N : n \rightarrow n' \wedge n' \rightarrow n'' \Rightarrow n \not\rightarrow n''$.*

   Structural edges point to the components of a composite data structure, composing the inner skeleton of its abstract syntax tree. More precisely, each field in a data type is represented with a separate node that is connected to the PDG node that contains the composite data structure. For instance, the structural edges of the CE-PDG in Figure 4.3 represent the tuples of the code in Figure 4.2. The second condition of the definition enforces the tree structure as otherwise "transitive" edges could be established. For example, without the second condition a structural edge between `{[C],D} = Z` and `C` could exist.

2. **Flow dependence identification**. The second step is to identify the flow dependences that arise from the decomposition of the data structure. Clearly, the new nodes can be variables that flow-depend on other nodes, so we need to identify the flow dependences that exist among the new (intra-statement) nodes. They can be classified according to two different scenarios: composite data structures being (i) defined and (ii) used. In Figure 4.2 we have a definition (line 4), a use (line 3) and a definition and use in the same node (line 2). The explicit definition of a whole

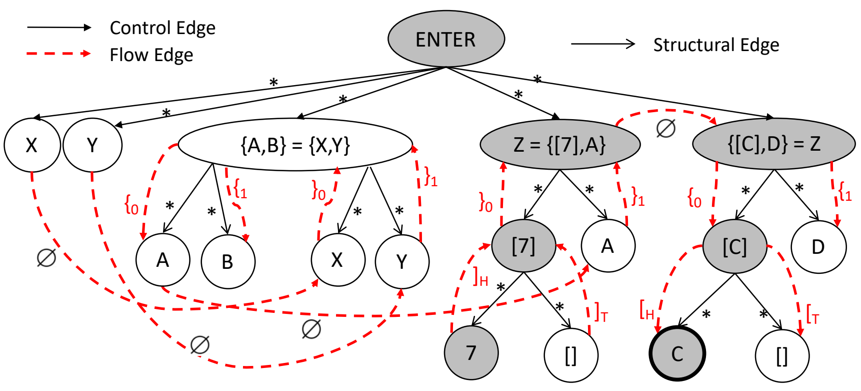


FIGURE 4.3: CE-PDG of the code in Figure 4.2

composite data structure (e.g., a tuple in the left-hand side of an assignment, see line 4) always defines every element inside it, so the values of all subelements depend on the structure that immediately contains them. Hence, the subexpressions depend on the structure being defined (i.e., flow edges follow the same direction as structural edges. See `{[C],D}=Z` in Figure 4.3). Conversely, the structure being used depends on its subexpressions (i.e., flow edges follow the opposite direction than structural edges. See `Z={[7],A}` in Figure 4.3). Additionally, because the decomposition of nodes augments the precision of the graph, all flow edges that pointed to original PDG nodes that have been decomposed, now point to the corresponding node in the new tree structure. An example of a flow edge that has been moved due to the decomposition is the flow edge between the new `A` nodes. In the original PDG, this flow edge linked the nodes `{A,B}={X,Y}` and `Z={[7],A}`.

3. **Edge labelling**. The last step to obtain the CE-PDG is labelling the edges with constraints that are used during the slicing phase. The idea is that the slicing algorithm traverses the edges and collects the labels in a stack that is used to decide what edges should be traversed and what edges should be ignored. We call the new labelled edges *constrained edges* because the labels act as constraints for the graph traversal.

Definition 4.2 (Constraint). *A constraint C is a label defined as follows:*

$$C ::= \emptyset \mid * \mid Tuple \mid List \qquad Pos ::= H \mid T$$
$$Tuple ::= \{_{int} \mid \}_{int} \qquad List ::= [_{Pos} \mid]_{Pos}$$

The meaning of each kind of constraint is the following:

- **Empty Constraint** ($n \xrightarrow{\emptyset} n'$). It specifies that an edge can always be traversed by the slicing algorithm.
- **Asterisk Constraint** ($n \xrightarrow{*} n'$). It also indicates that an edge can always be traversed; but it ignores all the collected restrictions so far, meaning that the whole data structure is needed.

- **Access Constraint ($n \xrightarrow{type_{position}} n'$).** It indicates that an element is the *position*-th component of another data structure that is a tuple if *type*=*Tuple* or a list if *type*=*List*. The *type* also indicates whether the element is being defined ({, [) or used (},]).

For the sake of simplicity, and without loss of generality, we distinguish between tuples and lists. The position in a tuple is indicated with an integer, while the position in a list is indicated with head (H) or tail (T). The case of objects, records, or any other structure can be trivially included by just specifying the position with the name of the field. Arrays where the position is a variable imply that any position of the array may be accessed. Hence, arrays with variable indices are treated as $\{_*$ constraints, which would match a constraint $\}_x$ for any x.

For instance, all edges in Figure 4.3 are labelled with constraints. Because `B` is the second element being defined in the tuple `{A,B}`, the constraint of the flow dependence edge that connects them is $\{_1$. Also, because 7 is the head in the list `[7]`, the constraint of the flow dependence edge that connects them is $]_H$.

At this point, it can be seen that the constraints can accurately slice the program in Figure 4.2. In the CE-PDG (Figure 4.3), the slicing criterion (`C`) is the head of a list (indicated by the constraint $[_H$), and this list is the first element of a tuple. When traversing backward the flow dependences, we do not want the whole `Z`, but the head of its first element (i.e., the cumulated constraints $[_H\{_0$). Then, when we reach the definition of `Z`, we find two flow dependences (`[7]` and `A`). But looking at their constraints, we exactly know that we want to traverse first $\}_0$ and then $]_H$ to reach the `1`. The slice computed in this way is composed of the grey nodes, and it is exactly the minimal slice in Figure 4.2b.

All edges in the CE-PDG are labelled with different constraints:

- Structural and control edges are always labelled with asterisk constraints.
- Flow edges for definitions are labelled with opening ({,[) access constraints.
- Flow edges for uses are labelled with closing (},]) access constraints.
- The remaining flow edges are labelled with empty constraints.

The CE-PDG is a generalisation of the PDG because the PDG is a CE-PDG where all edges are labelled with empty constraints ($\varnothing$) which have no effect over the graph traversal. The behaviour of access constraints and asterisk constraints in the graph traversal is further detailed in the next sections.

4.2 Dealing with recursive data structures

In this section we show that our technique is not only an alternative to atomisation but a fundamental improvement that solves an important problem that cannot be solved with atomisation. Atomisation cannot handle recursive data types such as lists, trees, linked lists (often implemented via a node that contains a value and a reference to another node), etc. The fundamental problem is that atomisation would need to infinitely unroll the recursive data type.

Example 4.2 (Recursive unfolding cannot be atomised)**.** *Consider the program in Figure 4.6c, in which a recursive tuple x is unfolded in a loop. The inside of the loop cannot be atomised, as the loop contains two instructions: $b = x$ and $\{x, a\} = b$. Atomisation would convert structures to multiple assignments. However, the depth of x and thus b is unknown:*

(1) $b = x; x = \mathbf{b_1}; a = \mathbf{b_2};$
(2) $b_1 = \mathbf{x_1}; b_2 = \mathbf{x_2}; x = b_1; a = b_2;$
(3) $b_1 = x_1; b_2 = x_2; x_1 = \mathbf{b_{11}}; x_2 = \mathbf{b_{12}}; a = b_2;$

A first step (1) is to split the pattern matching assignment into two separate assignments for x and a. However, b_1 and b_2 are not defined, so we must split b's assignment. The result (2) still has undefined variables (x_1 and x_2). Defining those would require splitting b_1 (3), resulting in a situation analogous to the first split, starting an infinite loop.

In contrast, the CE-PDG represents each field explicitly, performing the unfolding process as many times as required during the slicing traversal and thus, yielding to the correct slice.

4.3 Slicing the CE-PDG

The generation of new kinds of edges (structural edges) and the labelling introduced into the edges of the PDG makes the standard slicing algorithm (see Section 2.2 of Chapter 2) not suitable to compute program slices. The CE-PDG requires the constraints to be specifically treated during the traversal to calculate accurate program slices.

In order to represent the paths of the CE-PDG that can be traversed, we use a grammar. The label of an edge can be seen as a terminal. Therefore, by traversing the edges we build words. Unfortunately, not all edges can be traversed in any order; paths are only realizable when the word induced by the path belongs to a language for which the grammar is shown in Figure 4.4a. In this grammar, S is the initial symbol, C, R, and O represent sequences that contain closing, resolved, and opening constraints, respectively. $\varnothing$ and $*$ stand for empty and asterisk constraints, respectively. The key point of this grammar are resolved constraints. A resolved constraint is an opening constraint followed by *the complementary* closing constraint (e.g., $\{_2$ followed by $\}_2$).

Taking this into account, the grammar recognises paths formed by any combination of closing constraints followed by opening constraints. Any number of resolved constraints can be placed along the path. Because empty constraints ($\varnothing$) can always be traversed, they do not have any impact on the allowed paths. On the other hand, asterisk constraints ($*$) do have an impact because they always ignore any constraints already collected. Therefore, after traversing an

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

(A) Realisable paths grammar

$$
\begin{aligned}
S' &::= O' \\
O' &::= \{_i O' \mid [_p O' \mid \epsilon
\end{aligned}
$$

(B) Stack words

FIGURE 4.4: Grammars defining allowed constraints
($p \in \{H, T\}$ and $i \in \mathbb{Z}$)

asterisk constraint, the new traversable paths are the same as if no constraint was previously collected, hence they are followed by the initial symbol (S).

The semantic interpretation of the realisable paths grammar is the following: closing constraints along the path (C) indicate that we are traversing a use of a data structure. Opening constraints (O) indicate that we are only interested in the definition of an inner part of a data structure. Finally, resolved constraints (R) are associated to elements compressed into a variable and expanded again in a later statement (see lines 3 and 4 of the code in Figure 4.2a). For instance, consider variable `A` in the code of line 2 in Figure 4.2. Consider that we are interested in the value of this `A` variable. In this case, `A` is being defined in this line, so the value given to it cannot be inside the tuple being defined (the left-hand side of the equality). For this reason, we need to go up in the structure, until we exit from the structure where `A` is contained. Additionally, since `A` is inside a particular position of the composite structure (position 0 inside the tuple `{A,B}`), it can only receive a value of the same position in an analog composite structure. This may happen in the same statement or in any other previous one. For this reason, we metaphorically "open" a research process to find the value of variable `A`, and indicate it with the corresponding structure and position symbols ($\{_0$). When we reach the equality level, we notice that the expression giving value to the whole data structure is an analog data structure (the right-hand side of the equality, tuple `{X,Y}`). Thus, the expression that gives value to variable `A` must be inside this data structure. The problem is that this data structure includes more than one position. Fortunately, we are actively looking for a specific position with an open research ($\{_0$), and we can choose the element we are interested in between the possible ones ($\}_0$). We choose the correct element, reaching the variable that defines `A` and "closing" this specific research. As a result we reach a state where the pending research have been "resolved" and we can now focus on another open research if any. To sum up, with this point of view, each variable definition contained in a composite structure is considered as the opening of a flow dependence research, and each variable use inside an analogous structure as the closing of this flow dependence research. Then, the every time a research is successfully closed we say that the opened research has been resolved.

Example 4.3. *Consider Figure 4.3 and the slicing criterion (`C`). To compute the slice we trace a path from `C` to `1` formed by the following sequence of constraints: $[_H\{_0\varnothing\}_0]_H$, which can be derived with the grammar in Figure 4.4a:*

$$S \xrightarrow{S\to CO} CO \xrightarrow{C\to\epsilon} O \xrightarrow{O\to RO} RO \xrightarrow{O\to\epsilon} R \xrightarrow{R\to[_H R]_H} [_H R]_H \xrightarrow{R\to\{_0 R\}_0}$$
$$[_H\{_0 R\}_0]_H \xrightarrow{R\to\varnothing R} [_H\{_0\varnothing R\}_0]_H \xrightarrow{R\to\epsilon} [_H\{_0\varnothing\}_0]_H$$

The slicing algorithm uses a stack to store the words while it traverses the CE-PDG. When a node is selected as the slicing criterion, the algorithm starts from this node with an empty stack ($\perp$) and accumulates letters with each edge traversed. During the traversal, an edge can be traversed only if the word formed by collecting its symbol is accepted by Grammar 4.4a. Only opening constraints impose a restriction on the symbols that can be pushed onto the stack: when an opening constraint is on the top of the stack, the only closing constraint accepted to build a realizable word is its complementary closing constraint. Therefore, the only information necessary to determine whether an edge can be traversed is the sequence of non-resolved opening constraints at the top of the stack. They form the words that remain in the stack when a path is traversed (Grammar 4.4b).

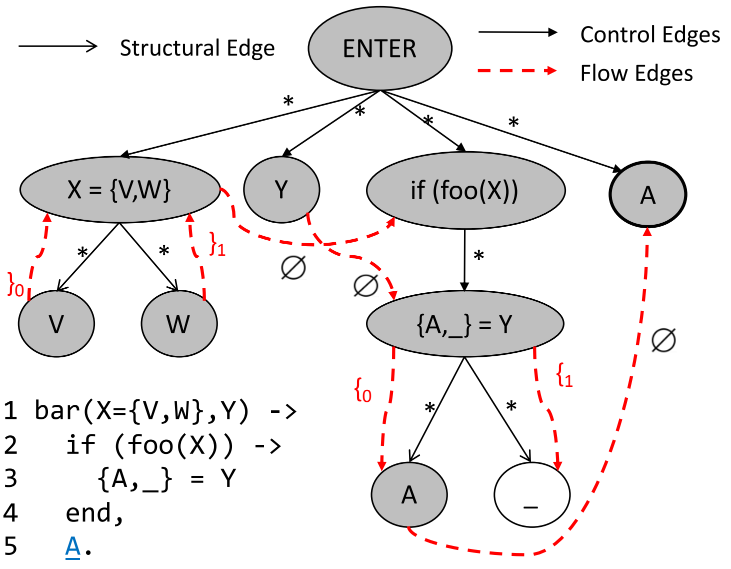
	Input Stack	Edge Constraint	Output Stack
(1)	S	$\varnothing$	S
(2)	S	$\{_x$ or $[_x$	$S\{_x$ or $S[_x$
(3)	$\perp$	$\}_x$ or $]_x$	$\perp$
(4)	$S\{_x$ or $S[_x$	$\}_x$ or $]_x$	S
(5)	$S\{_x$ or $S[_x$	$\}_y$ or $]_y$	*error*
(6)	S	$*$	$\perp$

TABLE 4.1: Processing edges' stacks. x and y are positions (*int* or H/T). $\varnothing$ and $*$ are empty and asterisk constraints, respectively. S is a stack, $\perp$ the empty stack.

Table 4.1 shows how the stack is updated in all possible situations. The constraints are collected or resolved depending on the last constraint added to the word (the one at the top of the *Input stack*) and the new one to be treated (column *Edge Constraint*). All cases shown in Table 4.1 can be summarised in four different situations:

- **Traverse constraint (cases 1 and 3):** The edge is traversed without modifying the stack.
- **Collect constraint (case 2):** The edge can be traversed by pushing the edge's constraint onto the stack.
- **Resolve constraint (cases 4 and 5):** There is an opening constraint at the top of the stack and an edge with a closing constraint that matches it (case 4), so the edge is traversed by popping the top of the stack; or they do not match (case 5), so the edge is not traversed.
- **Ignore constraints (case 6):** Traversing the edge empties the stack.

Example 4.4. *This example complements previous examples showing the use of asterisk constraints, empty constraints, and opening (access) constraints that* *are not resolved during the traversal. Consider the Erlang function and its associated CE-PDG in Figure 4.5, where variable `A` in line 5 is the slicing criterion and function `foo` is any boolean function (it can be ignored). Consider also the table in Figure 4.5, where the backward traversal of the graph is shown step by step (only the most relevant steps are shown). Each row represents the traversal of one edge, except the initial row that represents the initial node (the slicing criterion). In the table, column **Step** represents the number of the current step during the traversal. Different alternative paths are shown with letters (a,b,c). Column **NodeReached** represents the node reached after traversing the current edge, **EdgeType** and **EdgeConstraint** represent the type and constraint of the traversed edge, respectively; and **Stack** represents the stack computed after traversing the edge.*



```
1 bar(X={V,W},Y) ->
2   if (foo(X)) ->
3     {A,_} = Y
4   end,
5   A.
```

Step	NodeReached	EdgeType	EdgeConstraint	Stack
0	(5,`A`)	-	-	$\perp$
1	(3,`A`)	flow	$\emptyset$	$\perp$
2	(3,`{A,_} = Y`)	flow	$\{_0$	$\{_0$
3a	(1,`Y`)	flow	$\emptyset$	$\{_0$
4a	(1,ENTER)	control	*	$\perp$
3b	(2, `if(foo(X))`)	control	*	$\perp$
4b	(1,`X={V,W}`)	flow	$\emptyset$	$\perp$
5b	(1,`V`)	flow	$\}_0$	$\perp$
5c	(1,`W`)	flow	$\}_1$	$\perp$

FIGURE 4.5: Erlang function, associated CE-PDG, and slice step by step

After reaching `{A,_} = Y` (step 2), the stack contains the opening constraint $\{_0$, and there are two possible paths: (a) a flow path to the parameter variable `Y` (step 3a), and (b) a control path to the `if` condition (step 3b). Let us focus on the second path to show the necessity of asterisk constraints. When we traverse the control edge, all the constraints stacked due to the traversal of previous flow edges must be dropped from the stack (case 6 in Table 4.1). The reason is simple: when we reach a statement by a control edge, we are no longer interested *in the value of the uses of variables that the traversal has accumulated in the stack, but in the value of the variables used in this controller statement. The fact is that keeping the previous stack constraints may result in erroneous slices. For instance, consider a scenario where, if we do not empty the stack in step 3b, we would reach the* `X={V,W}` *statement with the stack* $\{_0$*, and the traversal would only reach the first element of the tuple (*V*) traversing* $\}_0$*. Therefore,* W *would be never included in the slice because it can only be reached traversing the constraint* $\}_1$ *that does not match the constraint of the stack. In contrast, emptying the stack in step 3b when traversing the control edge forces the slice to correctly include both* V *and* W*. Note also that the constraint* $\{_0$ *collected in step 2 is not entirely useless. It is still used in the flow path to* Y *(we only want the first component of* Y*).*

Algorithm 4.1 illustrates the process to slice the CE-PDG. It works similar to the standard algorithm [169], traversing backwards all edges from the slicing criterion and collecting nodes to form the final slice. The algorithm uses a work list with the states that must be processed. A state represents the (backward) traversal of an edge. It includes the node reached, the current stack, and the sequence of already traversed edges (line 6). In every iteration the algorithm processes one state. First, it collects all edges that target the current node (function getIncomingEdges in line 7). If the previous traversed edge is structural, we avoid traversing flow edges (lines 9–10) and only traverse structural or control dependence edges. The reason for this is that structural edges are only traversed to collect the structure of a data type so that the final slice is syntactically correct (for instance, to collect the tuple to which an element belongs). Flow edges are not further traversed to avoid collecting irrelevant dependences of the structural parent. Function processConstraint checks the existence of a potential traversal loop (when the traversal reaches an already traversed edge with a different stack) during the slicing phase and implements Table 4.1 to produce the new stack generated by traversing the edge to the next node (line 11). If the edge cannot be traversed according to Table 4.1 ($newStack$ == $error$), then the reachable node is ignored (line 12). Otherwise, the node is added to the work list together with the new stack (line 13). Finally, the state is added to a list of processed states, used to avoid the multiple evaluation of the same state, and the current node is included in the slice (lines 14–16).

Function processConstraint computes a new stack for all possible types of constraint: First, it returns an empty stack for asterisk constraints (line 20), Then, the condition in line 22 checks the existence of a loop (reaching an already traversed edge) during the slicing traversal. Function findLoop (line 23) returns the shortest suffix of the sequence of traversed edges that form the last loop, while function isIncreasingLoop (line 23), whose rationale is extensively explained in the next section, consequently empties the stack when needed. If no dangerous loop is detected, the function returns the same stack for empty constraints (line 24), or it processes access constraints following Table 4.1 with function processAccess (line 25).

Let's see how Algorithm 4.1 operates with the example code of Figure 4.2.

Algorithm 4.1 Intraprocedural slicing algorithm for CE-PDGs

Input: The slicing criterion node n_{sc}.
Output: The set of nodes that compose the slice.

```
1: function slicingAlgorithmIntra($n_{sc}$)
2:   $slice \leftarrow \emptyset; processed \leftarrow \emptyset$
3:   $workList \leftarrow \{\langle n_{sc}, \perp, [] \rangle\}$
4:   while $workList \neq \emptyset$ do
5:     select some $state \in workList$;
6:     $\langle node, stack, traversedEdges \rangle \leftarrow state$
7:     for all $edge \in$ getIncomingEdges($node$) do
8:       $\langle sourceNode, type, \_ \rangle \leftarrow edge$
9:       if getLastEdgeType($traversedEdges$) $= structural \wedge type = flow$ then
10:        continue for all
11:      $newStack \leftarrow$ processConstraint($stack$, $edge$)
12:      if $newStack \neq error$ then
13:        $workList \leftarrow workList \cup \{\langle sourceNode, newStack, traversedEdges$ ++ $edge \rangle\}$
14:    $processed \leftarrow processed \cup \{state\}$
15:    $workList \leftarrow \{(n, s, t) \in workList \mid (n, s, \_) \notin processed\}$
16:    $slice \leftarrow slice \cup \{node\}$
17:  return $slice$

18: function processConstraint($stack$, $edge$)
19:  $\langle \_, \_, constraint \rangle \leftarrow edge$
20:  if $constraint = AsteriskConstraint$ then return $\perp$
21:  else
22:    if $edge \in traversedEdges$ then
23:      if isIncreasingLoop(findLoop(traversedEdges),edge) then return $\perp$
24:    if $constraint = EmptyConstraint$ then return $stack$
25:    else return processAccess($stack$, $constraint$)

26: function processAccess($stack$, $constraint = \langle op, position \rangle$)
27:  if $stack = \perp$ then
28:    if $op = \{ \vee op = [$ then return $[constraint]$
29:    else return $\perp$
30:  $lastConstraint \leftarrow top(stack)$
31:  if $(op = \} \wedge lastConstraint = \langle \{, position \rangle)$
        $\vee (op = ] \wedge lastConstraint = \langle [, position \rangle)$ then
32:    pop($stack$)
33:  else
34:    if $op = \} \vee op = ]$ then return $error$
35:    else push($constraint, stack$)
36:  return $stack$
```

Consider again function `foo` in the code of the figure, the selected slicing criterion ($\langle 4, \texttt{C}\rangle$), and its CE-PDG, shown in Figure 4.3. The slicing process starts from the node that represents the slicing criterion (the expanded representation of the CE-PDG allows us to select `C`, the bold node, inside the tuple structure, excluding the rest of the tuple elements). Algorithm 4.1 starts the traversal of the graph with an empty stack ($\perp$). The evolution of the stack after traversing each flow edge is the following:

$$\perp \xrightarrow{[_H} [_H \xrightarrow{\{_0} [_H\{_0 \xrightarrow{\varnothing} [_H\{_0 \xrightarrow{\}_0} [_H \xrightarrow{]_H} \perp$$

Due to the limitations imposed by the grammar (and particularly by row 5 in Table 4.1), node `A` is never included in the slice because the following transition is not possible:

$$[_H\{_0 \xrightarrow{\}_1} error$$

Note that the slicing algorithm will also traverse the structural edges reaching the traversed nodes and generate new states in the worklist with empty stacks due to the asterisk constraint, however, no flow dependence edge is traversed after a structural edge and therefore despite the node `Z={[7],A}` being encountered with an empty stack, the structural edge to `A` is not traversed. As already noted, the resulting slice provided by Algorithm 4.1 is exactly the minimal slice shown in Figure 4.2b.

Slicing flow dependence loops

In static slicing we rarely know the values of variables (they often depend on dynamic information), so we cannot know how many iterations will be performed in a loop[1] (see the programs in Figure 4.6, where the value of `max` is unknown). For the sake of completeness, we must consider any number of iterations, thus program loops are often seen as potentially infinite. Program loops produce cycles in the PDG. Fortunately, the traversal of cycles in the PDG is not a problem, since every node is only visited once. In contrast, the traversal of a cycle in the CE-PDG could produce a situation in which the stack grows infinitely (see Figure 4.6c[2]), generating an infinite number of states. Fortunately, not all cycles produce this problem:

To keep the discussion precise, we need to formally define when a cycle in the CE-PDG is a *loop*.

Definition 4.3 (Loop)**.** *A cyclic flow dependence path* $P = n_1 \xleftarrow{C_1} n_2 \ldots \xleftarrow{C_n} n_1$ *is a* loop *if* P *can be traversed* $n > 1$ *times with an initial empty stack* ($\perp$) *following the rules of Table 4.1.*

[1]Note the careful wording in this section, where we distinguish between "program loops" (`while`, `for`...), "cycles" (paths in the PDG that repeat a node), and "loops" (repeated sequence of nodes during the graph traversal).

[2]It is easier to see how the stack changes by reading the code backwards from the slicing criterion.

```
read(max);
read(b);
x = init_tuple();
for(int i=0; i>max; i++){
  a = {x,i};
  x = {a,b};
}
{c,d} = x;
print(c);
```

(A) Decreasing stack size

```
read(max);
x = init_tuple();
for(int i=0; i>max; i++){
  {a,b} = x;
  x = {a+i,b+i};
}
{c,d} = x;
print(c);
```

(B) Same stack size

```
read(max);
x = init_tuple();
for(int i=0; i>max; i++){
  b = x;
  {x,a} = b;
}
{c,d} = x;
print(c);
```

(C) Increasing stack size

```
read(max);
x = init_tuple();
for(int i=0; i>max; i++){
  {e,d} = x;
  {a,b} = d;
  x = {a,b};
}
{c,d} = x;
print(c);
```

(D) Increasing stack size

FIGURE 4.6: Slicing looped data dependences in the CE-PDG (slicing criterion underlined and blue, sliced code in gray)

There exist three kinds of loops:

1. Loops that decrease the size of the stack in each iteration (Figure 4.6a) can only produce a finite number of states because the stack will eventually become empty. Such loops can be traversed collecting the elements specified by the stack, without a loss of precision.

   Definition 4.4 (Decreasing loop)**.** *A loop L is a* decreasing loop *if the number of closing constraints along L is greater than the number of opening constraints.*

2. Loops that repeat the same stack over and over after the first iteration (Figure 4.6b) are also not a problem because traversing the loop multiple times does not generate new states. Again, they can be traversed as many times as required by the stack, without a loss of precision.

   Definition 4.5 (Balanced loop)**.** *A loop L is a* balanced loop *if the number of closing and openings constraints along L is the same.*

3. Loops that increase the size of the stack in each iteration (Figure 4.6c) could produce an infinite number of states because the stack grows infinitely. However, not all increasing loops are dangerous. It is important to remark that not all cycles formed from more opening constraints than closing constraints are increasing loops. They may not even be loops (see

Definition 4.3). Cycles that are not loops are not dangerous because the cycle's edges constraints prevent us to traverse them infinitely. One illustrative example is the code in Figure 4.6d (see also its CE-PDG in Figure 4.7) where we have the flow dependence cycle:

$$(6,x) \xleftarrow{\}_0} (6,a) \xleftarrow{\emptyset} (5,a) \xleftarrow{\{_0} (5,d) \xleftarrow{\emptyset} (4,d) \xleftarrow{\{_1} (4,x) \xleftarrow{\emptyset} (6,x)$$

But this is not a loop because no matter with what stack we enter the cycle, when $\{_1$ is pushed on the stack, the cycle cannot be entered again due to the constraint $\}_0$ that does not match the top of the stack. In contrast, in the same code there exist a loop (highlighted in solid bold red) that can infinitely increase the stack with $\{_1$ in each iteration:

$$(6,x) \xleftarrow{\}_1} (6,b) \xleftarrow{\emptyset} (5,b) \xleftarrow{\{_1} (5,d) \xleftarrow{\emptyset} (4,d) \xleftarrow{\{_1} (4,x) \xleftarrow{\emptyset} (6,x)$$

We formally define a special kind of loop which is the only potentially dangerous: *increasing loop*.

Definition 4.6 (Increasing loop). *A loop L is an* increasing loop *if the number of opening constraints along L is greater than the number of closing constraints.*

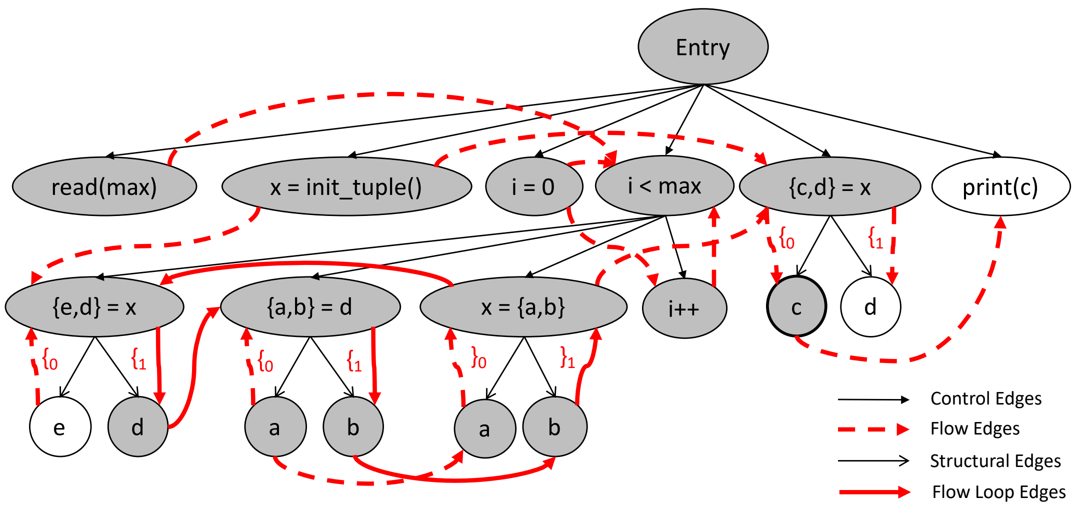


FIGURE 4.7: CE-PDG of Figure 4.6d. The increasing loop is represented in solid bold red

As we have seen, detection of increasing loops is not a trivial problem due to those flow cycles that are not loops. The pushdown automaton (PDA) of Figure 4.8 formalises (and detects) those loops that can grow the stack infinitely. The input of this automaton is the sequence of constraints that form a dependence loop. The PDA contains two states and two different stacks (closing stack and opening stack). Initial state 0 represents the case where all opening constraints of the sequence are balanced by the corresponding closing constraint. When a closing constraint is reached, the PDA pushes the constraint into the closing stack ($push_c$). When an opening constraint is processed, the PDA moves to state 1. Final state 1 represents the case where an opening constraint has been processed but not balanced yet. The transition to state 1 pushes the opening constraint into the opening stack $push_o$. In state 1, when a closing constraint that matches a previous opening constraint (condition M_o) is processed, we pop the opening constraint from the stack (pop_o). If the popped element of the opening stack is the last element of the stack (condition E_o), the PDA returns to state 0. Finally, if a path is accepted by this automaton, the path forms an increasing loop if and only if the inverted stack S_c is a prefix of S_o and they are not equal.

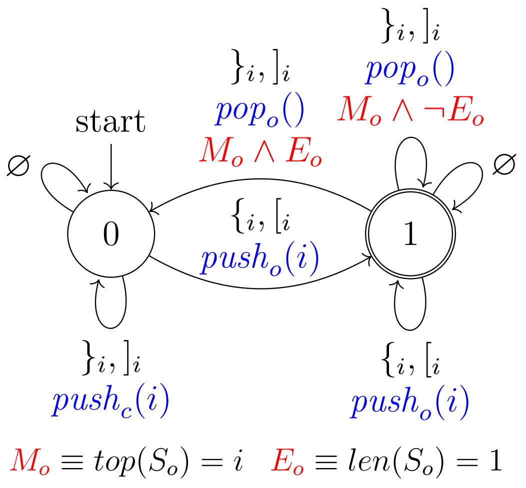


FIGURE 4.8: Pushdown automaton to recognize infinitely increasing loops

Note that * constraints do not appear in the PDA because they cannot appear in an increasing loop (an * constraint empties the stack and thus the same state would be repeated).

To properly understand how Algorithm 4.1 detects and solves loops, we show examples with each one of the situation given by flow loops of Figure 4.6.

Example 4.5 (Decreasing loop of Figure 4.6a). *Consider the code in Figure 4.6a and the slicing criterion $\langle 8, c\rangle$. If we traverse the CE-PDG of the program, shown in Figure 4.9, backwards from $(8, c)$ we find the following sequence of constraints*

$$(8,c) \xleftarrow{\{_0} (8,x) \xleftarrow{\varnothing} (6,x) \xleftarrow{\}_0} (6,a) \xleftarrow{\varnothing} (5,a) \xleftarrow{\}_0} (5,x) \xleftarrow{\varnothing} (6,x) \xleftarrow{\}_0} (6,a)$$
$$\xleftarrow[state]{repeated} STOP$$

with the following stack state for each step:

$$\perp \xleftarrow[c_8 \leftarrow x_8]{\{_0} \{_0 \xleftarrow[x_8 \leftarrow x_6]{\varnothing} \{_0 \xleftarrow[x_6 \leftarrow a_6]{\}_0} \perp \xleftarrow[a_6 \leftarrow a_5]{\varnothing} \perp \xleftarrow[a_5 \leftarrow x_5]{\}_0} \perp \xleftarrow[x_5 \leftarrow x_6]{\varnothing} \perp \xleftarrow[x_6 \leftarrow a_6]{\}_0} \perp$$

In this sequence, the edge $(6,x) \xleftarrow{\}_0} (6,a)$ is traversed twice with two different stacks ($\{_0 \neq \perp$). For this reason, the sequence of constraints of the loop

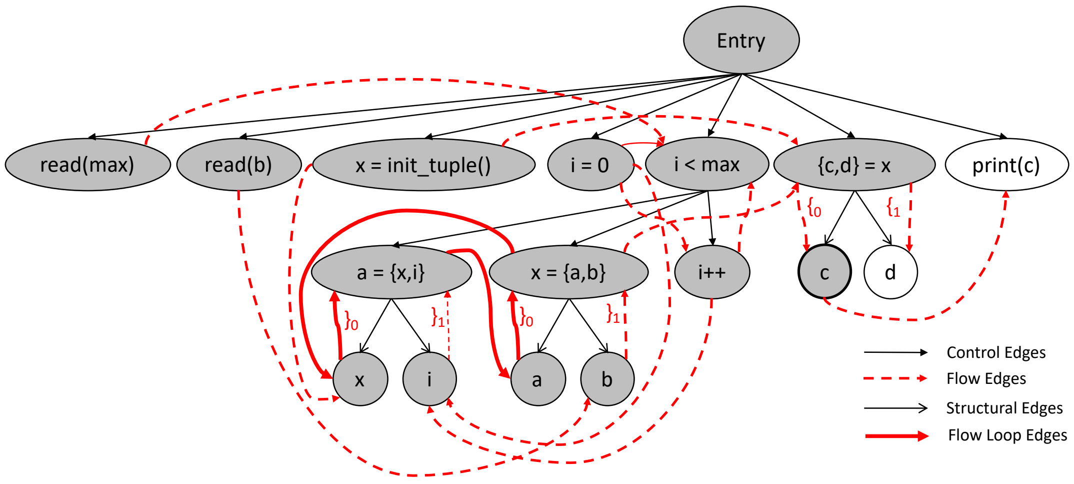


FIGURE 4.9: CE-PDG of Figure 4.6a

$(\}_0, \varnothing, \}_0, \varnothing)$ must be processed by the PDA of Figure 4.8 detecting whether the loop is an increasing loop or not. The trace of the PDA is shown in Figure 4.10.

$$\begin{tikzcd}
\boxed{\begin{array}{l} state = 0 \\ S_o = \bot \\ S_c = \bot \end{array}} \arrow[r, "\}_0"] & \boxed{\begin{array}{l} state = 0 \\ S_o = \bot \\ S_c = 0 \end{array}} \arrow[r, "\varnothing"] & \boxed{\begin{array}{l} state = 0 \\ S_o = \bot \\ S_c = 0 \end{array}} \arrow[r, "\}_0"] & \boxed{\begin{array}{l} state = 0 \\ S_o = \bot \\ S_c = 0,0 \end{array}} \arrow[r, "\varnothing"] & \boxed{\begin{array}{l} state = 0 \\ S_o = \bot \\ S_c = 0,0 \end{array}}
\end{tikzcd}$$

FIGURE 4.10: PDA trace for the loop of Figure 4.6a

As the final state is not an accepting state, the analysis by the PDA confirms that this is not an increasing loop, and the slicing traversal continues normally. The second time we traverse the edge $(6, x) \xleftarrow{\}_0} (6, a)$, we reach node $(6, a)$ with a repeated stack $(\bot)$ and the continuous loop traversal stops.

As can be seen, after the first iteration, the stack becomes empty, and the next iterations do not modify the stack. When the stack is empty, no restrictions exist in the graph traversal, and thus all edges are traversed. Therefore, at the end, all edges in the `for`-loop would be traversed. This is not a loss of precision, but a property of the data structure. Observe that $(5, i)$ and $(6, b)$ do actually influence the slicing criterion.

Example 4.6 (Balanced loop of Figure 4.6b). *Consider the code in Figure 4.6b, its CE-PDG shown in Figure 4.11, and the slicing criterion $\langle 7, \texttt{c} \rangle$. We only consider the path that, from the slicing criterion, reaches the balanced loop and stays in it (marked in solid bold red in Figure 4.11). There are other exits in the loop that are not traversed in this example for simplicity.*

The traversal would be as follows:

$$(7, c) \xleftarrow{\{_0} (7, x) \xleftarrow{\varnothing} (5, x) \xleftarrow{\}_0} (5, a + i) \xleftarrow{\varnothing} (4, a) \xleftarrow{\{_0} (4, x) \xleftarrow{\varnothing} (5, x)$$
$$\xleftarrow[state]{repeated} STOP$$

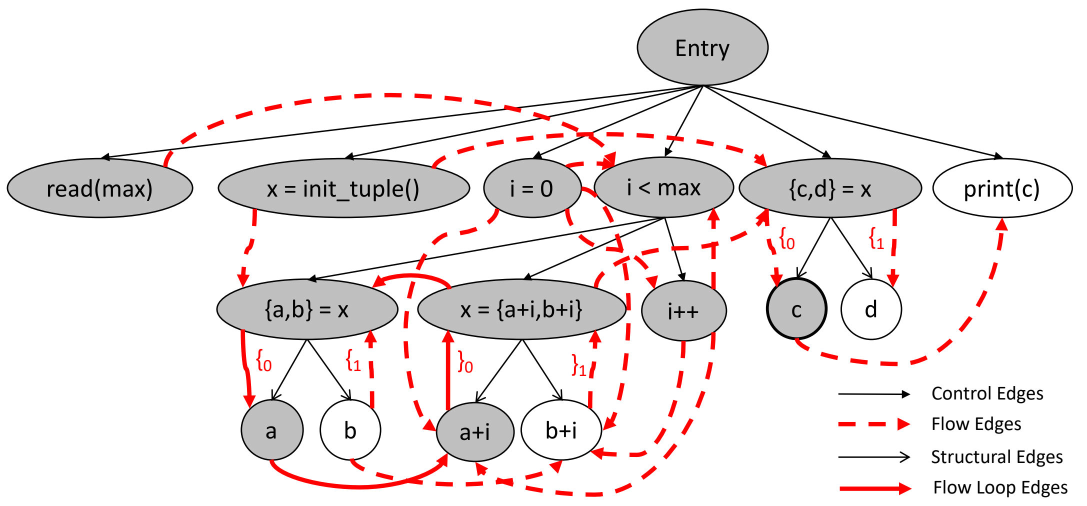


FIGURE 4.11: CE-PDG of Figure 4.6b

and this would be the state of the stack in each step:

$$\perp \xleftarrow[c_7 \leftarrow x_7]{\{_0} \{_0 \xleftarrow[x_7 \leftarrow x_5]{\varnothing} \{_0 \xleftarrow[x_5 \leftarrow a_5]{\}_0} \perp \xleftarrow[a_5 \leftarrow a_4]{\varnothing} \perp \xleftarrow[a_4 \leftarrow x_4]{\{_0} \{_0 \xleftarrow[x_4 \leftarrow x_5]{\varnothing} \{_0$$

Once $(5, x)$ *is reached again, the stack is the same* ($\{_0$), *so the edge* $(5, x) \xleftarrow{\}_0} (5, a+1)$ *is never traversed more than once, and thus the slicing algorithm never even detects a loop, as its detection mechanism relies on traversing the same edge twice.*

Example 4.7 (Increasing loop of Figure 4.6c)**.** *Consider the code in Figure 4.6c, its CE-PDG in Figure 4.12 and the slicing criterion* $\langle 7, \mathsf{c}\rangle$. *The CE-PDG of this code contains an increasing loop highlighted in solid bold red in Figure 4.12.*

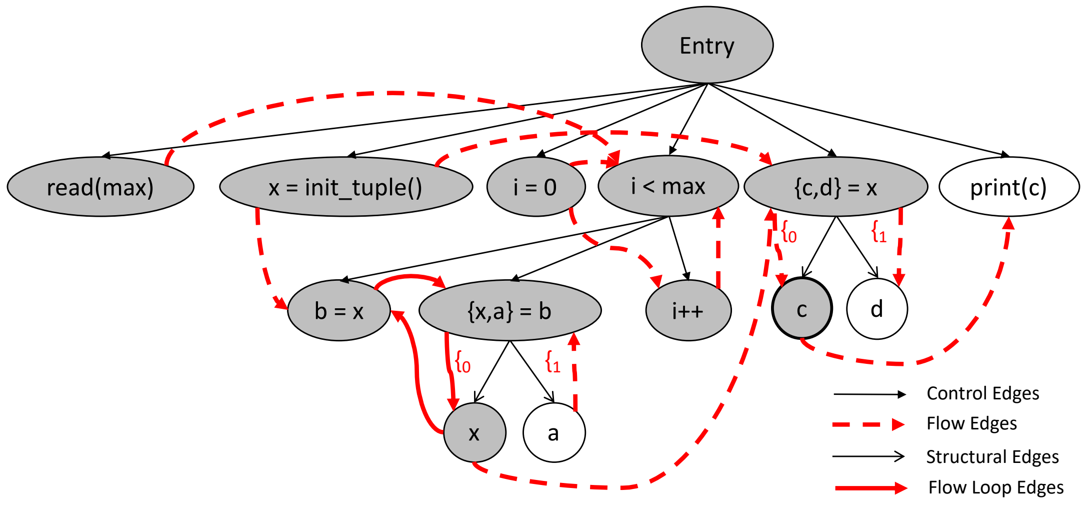


FIGURE 4.12: CE-PDG of Figure 4.6c

The following is the trace from the slicing criterion to the detection of the loop:

$$(8,c) \xleftarrow{\emptyset} (7,c) \xleftarrow{\{_0} (7,x) \xleftarrow{\emptyset} (5,x) \xleftarrow{\{_0} (5,b) \xleftarrow{\emptyset} (4,b=x) \xleftarrow{\emptyset} (5,x) \xleftarrow{\{_0} \dots$$

the stack has the following values throughout the trace:

$$\bot \xleftarrow[c_8\leftarrow c_7]{\emptyset} \bot \xleftarrow[c_7\leftarrow x_7]{\{_0} \{_0 \xleftarrow[x_7\leftarrow x_5]{\emptyset} \{_0 \xleftarrow[x_5\leftarrow b_5]{\{_0} \{_0\{_0 \xleftarrow[b_5\leftarrow b_4x_4]{\emptyset} \{_0\{_0 \xleftarrow[b_4x_4\leftarrow x_5]{\emptyset} \{_0\{_0 \xleftarrow[x_5\leftarrow b_5]{\{_0}$$

The edge $(5,x) \xleftarrow{\{_0} (5,b)$ has been traversed twice, as the stack with which $(5,x)$ was reached in the first pass is different from the one in the second pass ($\{_0 \neq \{_0\{_0$). Because an edge has been traversed twice, the PDA must be deployed, to check if this loop is an increasing loop.

The word to be fed into the automata is: $\{_0\emptyset\emptyset$; and the sequence of states would be:

$$\begin{tikzcd}
\boxed{\begin{array}{l} state = 0 \\ S_o = \bot \\ S_c = \bot \end{array}} \arrow[r, "\{_0"] & \boxed{\begin{array}{l} state = 1 \\ S_o = 0 \\ S_c = \bot \end{array}} \arrow[r, "\emptyset"] & \boxed{\begin{array}{l} state = 1 \\ S_o = 0 \\ S_c = \bot \end{array}} \arrow[r, "\emptyset"] & \boxed{\begin{array}{l} state = 1 \\ S_o = 0 \\ S_c = \bot \end{array}}
\end{tikzcd}$$

As the final state is an accepting state, the closing stack is a prefix of the opening stack, and $S_o \neq reverse(S_c)$, the analysis by the PDA confirms that this is an increasing loop. Therefore, the slicing algorithm empties the stack immediately, and proceeds to visit $(5,b)$ with an empty stack. This is denoted by $c \rightsquigarrow *$ (emptying the stack mimics the effect of traversing an $*$ constraint, which is the action performed instead of processing c, the constraint from the edge). The path from the slicing criterion would be:

$$\begin{aligned}(8,c) \xleftarrow{\emptyset} (7,c) \xleftarrow{\{_0} (7,x) \xleftarrow{\emptyset} (5,x) &\xleftarrow{\{_0} (5,b) \xleftarrow{\emptyset} (4,b=x) \xleftarrow{\emptyset} (5,x) \\ &\xleftarrow{\{_0} (5,b) \xleftarrow{\emptyset} (4,b=x) \xleftarrow{\emptyset} (5,x) \\ &\xleftarrow{\{_0} (5,b)\end{aligned}$$

and the stack follows this trace:

$$\begin{aligned}\bot \xleftarrow[c_8\leftarrow c_7]{\emptyset} \bot \xleftarrow[c_7\leftarrow x_7]{\{_0} \{_0 \xleftarrow[x_7\leftarrow x_5]{\emptyset} \{_0 &\xleftarrow[x_5\leftarrow b_5]{\{_0} \{_0\{_0 \xleftarrow[b_5\leftarrow b_4x_4]{\emptyset} \{_0\{_0 \xleftarrow[b_4x_4\leftarrow x_5]{\emptyset} \{_0\{_0 \\ &\xleftarrow[x_5\leftarrow b_5]{\{_0 \rightsquigarrow *} \bot \xleftarrow[b_5\leftarrow b_4x_4]{\emptyset} \bot \xleftarrow[b_4x_4\leftarrow x_5]{\emptyset} \bot \\ &\xleftarrow[x_5\leftarrow b_5]{\{_0 \rightsquigarrow *} \bot \xleftarrow[state]{repeated} STOP\end{aligned}$$

When the loop has been found, the traversal continues, and once the traversal reaches node $(5,b)$ for a third time, it stops, as two of the three stacks that reached it are the same ($\{_0\{_0$; $\bot$; $\bot$). Finally, the traversal through the loop stops.

Only increasing loops can produce non-termination. For this reason, Algorithm 4.1 detects loops (Line 22) and checks whether they are increasing with function isIncreasingLoop (Line 23). This function uses the PDA of Figure 4.8 to determine whether the loop is increasing. If it is increasing the stack is emptied, i.e., the traversal continues unconstrained. Since the traversal allows the traversal of the same edge several times when the stack is different, termination of the whole slicing process must be proved. Theorem 4.1 ensures termination of the whole slicing process.

To prove the theorem, we need to prove some properties of a loop that are needed. These properties are captured by the following lemmas.

Lemma 4.1. *Let L be a non-increasing loop and S be a stack. The traversal of L with S is terminating.*

Proof. Non-increasing loops can be decreasing loops (Definition 4.4) or balanced loops (Definition 4.5). We prove each case separately.

Decreasing loops. First of all, for any initial stack, a first traversal of all the edges of L must be possible to explore this scenario. Otherwise, there cannot be a traversal loop due to unmatched constraints. Moreover, according to Definition 4.4 the path of L contains more closing constraints than opening constraints. When slicing the graph, and before iterating into a decreasing loop, there are two stack possibilities: the empty stack or a stack with a sequence of opening constraints at the top (see Figure 4.4b):

Empty stack ($\perp$). Since the constraint in L can be in any order, we need to consider two different scenarios:

- *There is no suffix of L where the number of opening constraint is greater than the number of closing constraints (e.g., $\{_0\{_0\}_0\}_0\}_0$).* Since L is a decreasing loop, in this scenario we can summarise L as a sequence of n closing constraints. In this case, according to case 3 of Table 4.1, pushing any number j of closing constraint to an empty stack results in the empty stack.

$$\perp \xrightarrow{\}_{i,\forall i \in 1..j}}{}^{*} \perp \quad \text{and} \quad \perp \xrightarrow{]_{i,\forall i \in 1..j}}{}^{*} \perp$$

  Then, the same node is reached twice with the same stack ($\perp$) and a second traversal is not processed as indicated by line 15 in Algorithm 4.1.

- *There is a suffix of L where the number of opening constraints is greater than the number of closing constraints (e.g., $\}_0\}_0\{_0\{_0\}_0$).* In this scenario there are always closing constraints that are consumed by the initial empty stack ($\perp$) according to case 3 of Table 4.1. On the other hand, the final part of L pushes more opening constraints than closing constraints and the stack calculated after the first traversal of the loop is not empty. As L is a decreasing loop, all the constraints pushed in the first traversal are consumed at the beginning of the second traversal. Finally, the final part of L generates the same sequence of positive constraints at finishing the second traversal. Therefore, as the same nodes is reached twice with the same stack, the traversal stops.

$$\perp \xrightarrow{L} S \xrightarrow{L}{}^{*} S$$

Stack with a sequence of opening constraints $(C_{O_1} \dots C_{O_n})$. Cases 4 and 5 of Table 4.1 represent the two possibilities that can occur when pushing a closing constraint to a non-empty stack: the closing constraint balances the opening constraint at the top of the stack or it fails.

- According to case 5, if the closing constraint does not balance the opening constraint at the top of the stack, the traversal is aborted by an error. Hence, the traversal is finite.

$$\dots \{_i \xrightarrow{\}_j, \forall i \neq j} \mathit{error} \text{ and } \dots [_i \xrightarrow{]_j, \forall i \neq j} \mathit{error}$$

- On the contrary, case 4 shows that if the closing constraint balances the opening constraint at the top of the stack, the constraint is popped and the traversal continues. Since the number of elements of the stack is finite (an infinite stack would have been forever growing in an increasing loop), popping each opening constraint on the top of the stack with a closing constraint of the loop sequence will result in an empty stack at some point of the traversal. Note that this point of the traversal is not necessarily after traversing the whole loop L, but can be after traversing any intermediate edge of the loop. As the processing of any decreasing loop L with an empty stack $(\perp)$ has already been proved terminating, then we can state that the traversal is finite.

$$\{_1 \dots \{_n \xrightarrow{L}{}^{m} S \xrightarrow{L}{}^{*} S$$

Balanced loops. With the same reasoning as in the decreasing loops, we differentiate two scenarios according to the initial form of the stack:

Empty stack $(\perp)$. Since the constraint in L can be in any order, we need to consider the same two scenarios we did before:

- *There is no suffix of L where the number of opening constraint is greater than the number of closing constraints (e.g.,* $\{_0\{_0\}_0\}_0$*).* Since L is a balanced loop, in this scenario we can summarise L as an empty sequence of constraints. The result of iterating into a balanced loop of this type is the same stack, since each iteration of the loop itself balances the opening constraints with their complementary closing constraints. Hence, the initial node of the graph will be reached twice with the same stack, and the traversal will be stopped as indicated by line 15 in Algorithm 4.1.

$$\perp \xrightarrow{\varnothing}{}^{*} \perp$$

- *There is a suffix of L where the number of opening constraints is greater than the number of closing constraints (e.g., $\}_0\}_0\{_0\{_0$).* In this scenario there are always closing constraints that are consumed by the initial empty stack ($\perp$) according to case 3 of Table 4.1. On the other hand, the final part of L pushes more opening constraints than closing constraints and the stack calculated after the first traversal of the loop is not empty. As L is a balanced loop, all the constraints pushed in the first traversal are consumed at the beginning of the second traversal. Finally, the final part of L generates the same stack when the traversal of L finishes. Therefore, as the same node is reached twice with the same stack, the traversal stops.

$$\perp \overset{L}{\to} S \overset{L}{\to}^{*} S$$

Stack with a sequence of opening constraints ($C_{O_1} \dots C_{O_n}$). Cases 4 and 5 of Table 4.1 represent the two possibilities that can occur when pushing a closing constraint to a non-empty stack: the closing constraint balances the opening constraint at the top of the stack or it fails.

- According to case 5, if the closing constraint does not balance the opening constraint at the top of the stack, the traversal is aborted by an error. Hence, the traversal is finite.

$$\dots\{_i \xrightarrow{\}_j, \forall i \neq j} \textit{error} \text{ and } \dots [_i \xrightarrow{]_j, \forall i \neq j} \textit{error}$$

- On the contrary, case 4 shows that if the closing constraint balances the opening constraint at the top of the stack, the constraint is popped and the traversal continues. Since the number of opening and closing constraints in L is the same, if any constraint from the stack is consumed by the initial closing constraints it will be restored later by the corresponding opening constraint contained in L. Then, the stack obtained after traversing L will be the same. As a result, since the same node is reached twice with the same stack, the traversal finishes.

$$\{_1 \dots \{_n \overset{L}{\to}^{*} \{_1 \dots \{_n$$

Finally, note that if we consume every opening constraint during the traversal of the closing constraints at the beginning of L, we will find ourselves in the previous scenario where the stack was empty and, thus, the traversal finishes too.

$\square$

Lemma 4.2. *Given a CE-PDG increasing loop L. There exists a stack S for which it is possible to infinitely traverse L with S.*

Proof. Following the same reasoning that we did in some particular decreasing loops, each iteration of an increasing loop can be summarised as a sequence of opening constraints of the form $\{_1 \ldots \{_n$ independently of the sequence of constraints. Case 2 of Table 4.1 shows that opening constraints can always be traversed independently of the top of the stack. For this reason, the loop can be infinitely traversed generating an infinite stack.

$$S \xrightarrow{\{_i}^* S \mathbin{+\!\!+} \{_1 \ldots \{_n \{_1 \ldots \{_n \ldots$$

□

Lemma 4.3. *Given a CE-PDG increasing loop L, a stack S that allows Algorithm 4.1 to iterate at least once into it, and a loop L' which corresponds to L but replacing any access constraint by an asterisk constraint, then L' is not an increasing loop and it is not possible to infinitely traverse L' with S.*

Proof. According to Definition 4.6, a cyclic flow dependence path is an increasing loop if the sequence of constraints generated by traversing it belongs to the language induced by the PDA in Figure 4.8. The PDA cannot reach the final state if an asterisk constraint exist in L, thus, it cannot be an increasing loop. Moreover, the traversal of an asterisk constraint described in case 6 of Table 4.1 always result in an empty stack ($\perp$). Therefore, if an asterisk constraint is included in an edge of L, then the second time that this edge is traversed the same node will be reached again with the same stack ($\perp$). Therefore, a second traversal is never done as indicated by line 15 in Algorithm 4.1. □

After proving these three lemmas, we have all the needed components to enunciate and prove Theorem 4.1.

Theorem 4.1 (Termination of slicing)**.** *Let $P = (N, E)$ be a CE-PDG and let $n_{sc} \in N$ be a slicing criterion for P. Algorithm 4.1 terminates when it slices P with respect to n_{sc}.*

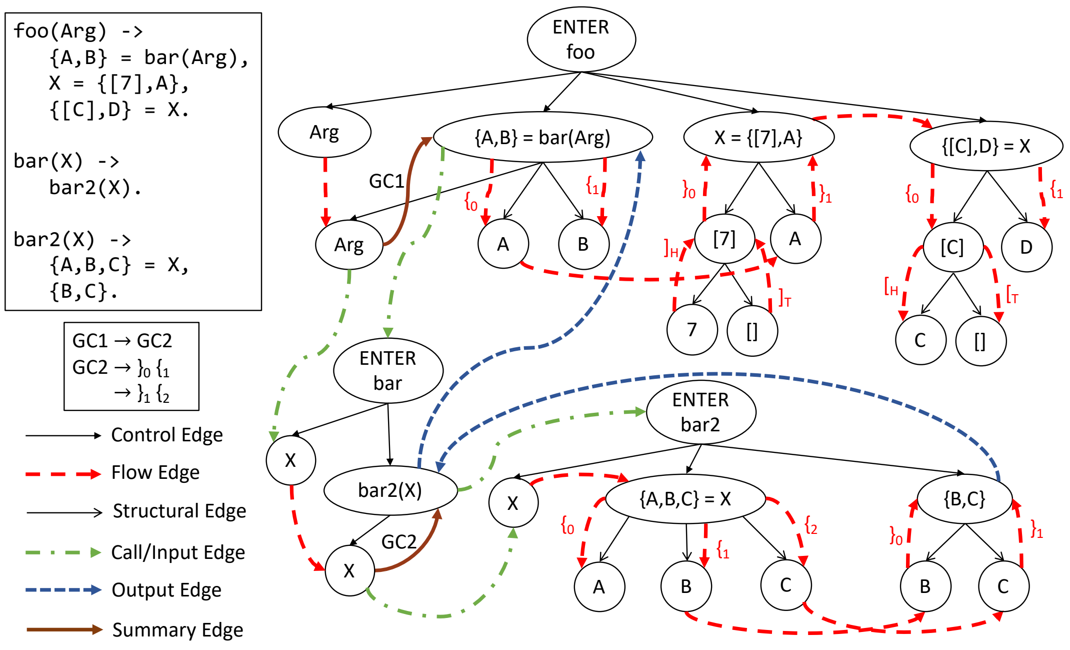
Proof. The traversal of any sequence of nodes that is not a cycle (i.e., that do not represent a loop in the program) trivially terminates. Only loops can produce non-termination in Algorithm 4.1. But all loops are detected by the algorithm in line 15. According to Lemma 4.1, the traversal of all non-decreasing loops always terminates. On the other hand, as shown in Lemma 4.2 increasing loops can produce non-termination. However, all of them are detected by the PDA in Figure 4.8. When an increasing loop is detected by the algorithm the stack is emptied.

We know by case 6 in Table 4.1 that including an asterisk constraint in a path is equivalent to emptying the stack. Therefore, according to Lemma 4.3 it is not possible to infinitely traverse the increasing loops found in the traversal made by the algorithm. Hence, Algorithm 4.1 always terminates. □

The reader could think that it would be a good idea to identify all increasing loops at CE-PDG construction time. Unfortunately, finding all loops has an average complexity $\mathcal{O}(N^2EL)$, where L is the number of loops. The worst complexity is exponential $\mathcal{O}(2^N)$ [71]. Our approach avoids the problem of finding all loops, determining whether they are increasing, and processing them. We treat them on demand, when they are found by the slicing algorithm (i.e., we do not search for loops, we just find them during the CE-PDG traversal). So we only process those loops found in the slicing traversal.

4.4 The CE-SDG

Function calls pose additional problems to the slicing of composite data structures. A data structure can be decomposed along various function calls so that the result of a function may depend only on a subexpression of its argument (and not on the rest). But, even worse, in presence of direct or transitive recursion, it may turn impossible to determine statically what parts of a function's argument do influence the result of that function.



```
foo(Arg) ->
   {A,B} = bar(Arg),
   X = {[7],A},
   {[C],D} = X.

bar(X) ->
   bar2(X).

bar2(X) ->
   {A,B,C} = X,
   {B,C}.
```

$GC1 \to GC2$
$GC2 \to \}_0 \{_1$
$\to \}_1 \{_2$

FIGURE 4.13: The interprocedural CE-SDG

Example 4.8. *Consider the code in Figure 4.13, which augments the example in Figure 4.2 to an interprocedural program. In this figure, tuple $\{X, Y\}$ in line 2 is replaced by a call to a method* `bar` *and arguments* `X` *and* `Y` *replaced by a single argument* `Arg`*.* `bar` *calls* `bar2`*, which uses the second and third elements of the* `X` *parameter tuple to compute the value that* `bar` *finally returns. Therefore, the returned value of the call to* `bar` *in Figure 4.13 only depends on the second and third elements of* `Arg`*. More important, if the slicing criterion is* `{A,B}` *in* `foo`*, then* `B` *and* `C` *in* `bar2` *should belong to the slice. However, if the slicing criterion is* `A` *in* `foo`*, then only* `B` *in* `bar2` *should belong to the slice (although the same edges in* `bar` *are traversed). This means that the dependences that are propagated not only depend on the program, but on the slicing criterion, which is unknown at the time of constructing the SDG. Clearly, we need a mechanism that traverses different functions propagating the information required by the* *slicing criterion for each data structure. This problem remains unsolved in program slicing.*

The problem shown in Example 4.8 can be solved with our stack, which propagates exactly the required information at slicing time thanks to the labelling of the edges introduced at CE-PDG construction time. The information must be propagated using the input and output edges that connect functions, and through summary edges, building a CE-SDG. We start by explaining why summary edges need a special treatment to properly represent whether the returned value of a function only depends on one or some specific parts of the argument. Then, we describe how summary edges must be labelled when the source code of the called function is not available; and we also explain the problem of dealing with recursive function calls.

4.4.1 Summary edges and grammar productions

In the SDG, the existence of a summary edge between a formal-in and a formal-out nodes means that the value of the formal-in can influence the value of the formal-out. This is calculated by just checking whether a dependence path exists between formal-in and formal-out nodes. Summary edges summarise that (or those) path(s).

However, in the CE-SDG, every dependence path between two nodes is represented with a stack that stores the sequence of constraints collected along the path. Therefore, a summary edge in the CE-SDG must indicate all possible stacks that it is summarising. Each stack represents a different dependence path between a formal-in and a formal-out.

Example 4.9. *Consider again the CE-SDG in Figure 4.13. The summary edge that represents the backward traversal paths between `bar2(X)` and `X` is labelled with `GC2` (Grammar Constraint 2). `GC2` represents two possible paths: $\}_0\{_1$ traversing node `B` and $\}_1\{_2$ traversing node `C`.*

In the rest of this section we explain how to compute the grammar constraints of summary edges and how to traverse them at slicing time.

Grammar constraints only appear in summary edges, so they are exclusive of interprocedural slicing. The grammar used to construct summary edges is shown in Figure 4.14a, which extends the one in Figure 4.4a with grammar constraints.

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

(A) Realisable path grammar

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

(B) Stack words

FIGURE 4.14: Allowed constraints when building summary edges ($p \in \{H, T\}$ and $i \in \mathbb{Z}$)

In interprocedural slicing, we distinguish two traversal processes: (i) the traversal performed to build summary edges (at CE-SDG construction time), and (ii) the traversal used to compute the slice (at slicing time). Although Grammar 4.14a represents the realisable paths for both processes, asterisk and grammar constraints are treated differently for each one of them. At summary construction time, asterisk, grammar, and closing access constraints are always pushed onto the stack (Grammar 4.14b), while at slice computation time they are resolved when traversed (Grammar 4.4b).

To create these new stack words, at summary construction time, the cases in Table 4.1 are extended with the ones in Table 4.2 (the changes are highlighted in blue italics).

	Input Stack	Edge Constraint	Output Stack
(1)	S	$\varnothing$	S
(2)	S	$\{_x$ or $[_x$	$S\{_x$ or $S[_x$
(3)	$\perp$	$\}_x$ or $]_x$	$\}_x$ or $]_x$
(4)	$S\{_x$ or $S[_x$	$\}_x$ or $]_x$	S
(5)	$S\{_x$ or $S[_x$	$\}_y$ or $]_y$	*error*
(6)	$S\}_x$ or $S]_x$	$\}_y$ or $]_y$	$S\}_x\}_y$ or $S]_x]_y$
(7)	$S\ *$	$\}_x$ or $]_x$	$S * \}_x$ or $S *]_x$
(8)	$S\ g$	$\}_x$ or $]_x$	$S\ g\ \}_x$ or $S\ g\ ]_x$
(9)	$S\ *\ S'$	$*$	$S\ *$
(10)	S	$*$	$S\ *$
(11)	S	g	$S\ g$

TABLE 4.2: Processing edges' stacks in the summary construction phase. x and y are positions (*int* or H/T). $\varnothing$, $*$, and g are empty, asterisk, and grammar constraints, respectively. S is a stack, $\perp$ the empty stack.

The computation of the summary edges in the CE-SDG is similar to the standard process (see, e.g., [169]). However, they are equipped with grammar constraints, whose information must replicate the behaviour of a function for all possible function calls. To achieve this, there are four main differences in our approach:

1. the traversal starts from a formal-out node with an empty stack ($\perp$) that is updated according to Table 4.2,

2. because edges are labelled with constraints, it is possible to not traverse an edge (see case 9 in Table 4.2),

3. one node can be processed several times if it is reached with different stack states, and

4. whenever a formal-in node is reached using a path, the obtained stack is stored to create one production of the grammar, and the rest of feasible paths are also explored.

Algorithm 4.2 is in charge of equipping the CE-SDG with summary edges and their grammar constraints. It is based on the summary detection algorithm presented in [169]. All the modifications of the algorithm focus on the constraints treatment. The algorithm starts with a work list that contains all states that must be processed. A state contains a pair of nodes, the current stack, and the last edge traversed to reach that state. At the beginning, the work list contains all formal-out nodes (Line 2), and it traverses backwards all edges from the nodes in the work list. Every time we traverse an edge, we process its constraint and include the reached node in the work list. If we reach a formal-in node (Line 7), then we have found a backward dependence path from the formal-out to the formal-in. Thus, we must add a new grammar production with the new path (Line 8) and, if there is not a summary edge in each call to the current procedure, we must include it (Line 9). On the other hand, if the node reached is not a formal-in node (Line 18), then we process (i.e., we traverse backwards) all incoming edges of the reached node. There are two situations in which the incoming edge must be ignored (Line 21):

- When the node reached is an actual-out node, only control and summary edges must be traversed. This avoids processing call out edges, which must be ignored to avoid descending into a procedure that has been already analysed and resumed by the corresponding summary edge.
- As in Algorithm 4.1, when a structural edge has been traversed, flow edges must be ignored.

The edges and the stack are processed according to Table 4.2 with function *processConstraint*, which is similar to the intraprocedural version in Algorithm 4.1. The only modification required is the one performed for asterisk, grammar, and closing access constraints, which are now always added to the stack (Lines 35-39).

Algorithm 4.2 Summary detection algorithm for CE-SDGs

Input: The CE-SDG without summary edges.
Output: The CE-SDG equipped with (constrained) summary edges and their associated grammar.

```
 1: function SUMMARYDETECTION(CE-SDG)
 2:   $workList \leftarrow \{\langle n \rightarrow n, \perp, null\rangle \mid n \in$ FORMALOUTNODES(CE-SDG)$\}$
 3:   $processed \leftarrow \emptyset$
 4:   while $workList \neq \emptyset$ do
 5:     select some $state \in workList$
 6:     $\langle n_s \rightarrow f_{out}, stack, traversedEdges\rangle \leftarrow state$
 7:     if $n_s \in$ FORMALINNODES(CE-SDG) then
 8:       $gConstraint \leftarrow$ ADDGRAMMARPRODUCTION($n_s$, $f_{out}$, $stack$)
 9:       for all $c \in$ CALLERS(PROC($n_s$)) do
10:         $a_{in} \leftarrow$ CORRESPONDINGACTUALIN($c$, $n_s$)
11:         $a_{out} \leftarrow$ CORRESPONDINGACTUALOUT($c$, $n_{f_{out}}$)
12:         if $\neg$EXISTSSUMMARYEDGE($a_{in}$, $a_{out}$) then
13:           $sumConstraint \leftarrow gConstraint$
14:           $summaryEdge \leftarrow$ ADDSUMMARYEDGE($a_{in}$, $a_{out}$, $sumConstraint$)
```

```
15:         for all $\langle a_{out} \to n, stack', traversedEdges' \rangle \in processed$ do
16:           $newStack \leftarrow$ PROCESSCONSTRAINT($stack'$, $summaryEdge$)
17:           $workList \leftarrow workList \cup \{\langle a_{in} \to n, newStack, traversedEdges' \cup summaryEdge \rangle\}$
18:     else
19:       for all $edge \in$ GETINCOMINGEDGES($n_s$) do
20:         $\langle n_1 \to n_s, type, \_ \rangle \leftarrow edge$
21:         if (GETLASTEDGETYPE($traversedEdges$)= $structural \wedge type = flow$) $\vee$
               ($n_s \in ActualOutNodes$(CE-SDG) $\wedge type \notin \{control, summary\}$) then
22:           continue for all
23:         $newStack \leftarrow$ PROCESSCONSTRAINT($stack$, $edge$)
24:         if $newStack \neq error$ then
25:           $workList \leftarrow workList \cup \{\langle n_1 \to f_{out}, newStack, traversedEdges$ ++ $edge \rangle\}$
26:     $processed \leftarrow processed \cup \{state\}$
27:     $workList \leftarrow workList \setminus processed$

28: function PROCESSCONSTRAINT($stack$, $edge$)
29:   $\langle \_, \_, constraint \rangle \leftarrow edge$
30:   if ISINCREASINGLOOP(FINDLOOP(traversedEdges),edge) then
31:     PUSH($constraint$, $stack$)
32:     return $stack$
33:   if $constraint = EmptyConstraint$ then
34:     return $stack$
35:   else if $constraint \in \{AsteriskConstraint, GrammarConstraint\}$ then
36:     PUSH($constraint$, $stack$)
37:     return $stack$
38:   else
39:     return PROCESSACCESS($stack$, $constraint$)

40: function PROCESSACCESS($stack$, $constraint = \langle op, type, position \rangle$)
41:   if $stack = \perp$ then
42:     return $[constraint]$
43:   $lastConstraint \leftarrow$ TOP($stack$)
44:   if ($op = \} \vee op = ]$) $\wedge$ ISACCESSCONSTRAINT($lastConstraint$) then
45:     $\langle lastOp, \_, \_ \rangle \leftarrow lastConstraint$
46:     if $lastOp = \} \vee lastOp = ]$ then
47:       PUSH($constraint$, $stack$)
48:     else if ($op = \} \wedge lastConstraint = \langle \{, position \rangle$)
               $\vee$ ($op = ] \wedge lastConstraint = \langle [, position \rangle$) then
49:       POP($stack$)
50:     else
51:       return $error$
52:   else
53:     PUSH($constraint$, $stack$)
54:   return $stack$
```

Example 4.10. *Consider the code and its associated CE-SDG shown in Figure 4.13. In this CE-SDG, asterisk constraints in control and structural edges, and empty constraints of flow edges have been omitted for simplicity. The summary edges of functions* `bar` *and* `bar2` *are labelled with the $GC1$ and $GC2$ grammar constraints, respectively. They represent the feasible slicing paths computed by Algorithm 4.2 from the formal-out to the formal-in nodes. We can use the LR(0) analysis (see Section "Items and the LR(0) Automaton" (pag. 242) in [3]) to traverse a grammar constraint with a stack. For instance, Figure 4.15* *shows the LR(0) automaton of the grammar constraint $GC1$. For clarity, the states (boxes) that can be reached through edges labelled with non-terminals are in grey (because they can be ignored).*

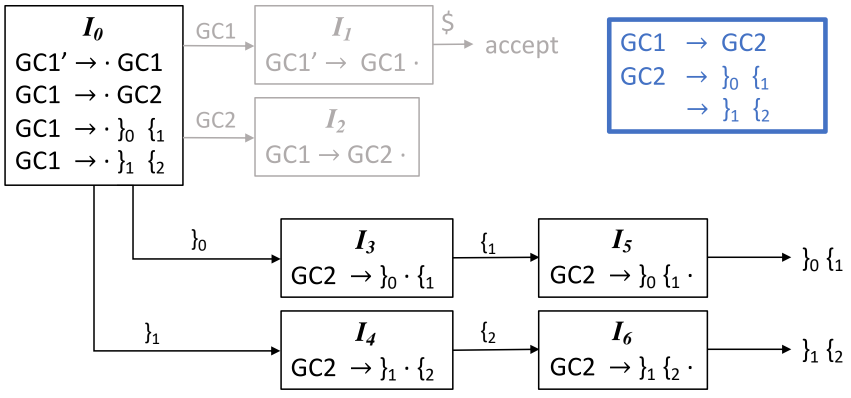


FIGURE 4.15: LR(0) automaton to represent a grammar constraint

Each box represents a step of the derivation. The dot written in each grammar production separate the already processed and pending elements of that production. In this example, there are only two paths, thus, this grammar constraint can produce two new stacks. Therefore, traversing the summary edge with the empty stack $\perp$ produces two different stacks: $\}_0\{_1$ and $\}_1\{_2$. Traversing the summary edge with the stack $\{_0$ only produces one stack: $\{_1$ because the other path produces error according to Table 4.2.

4.4.2 Summary constraints for unknown source code functions

When we deal with calls to functions whose code is unavailable (e.g., because they belong to precompiled libraries), there is no information about the function being called. Therefore, it remains unknown what part of the argument is needed to compute the result.

Example 4.11. *Consider the code in Figure 4.16, where the code of function `gee` is missing. Therefore, it is not possible to analyse how the returned value `{A,B}` depends on argument `Arg`. E.g., if `Arg` is a tuple, then it is impossible to know which positions of the tuple influence `A` or `B`.*

```
main(Arg) ->
  {A,B} = gee(Arg),
  A.
```

FIGURE 4.16: Erlang program with an call to an unknown function

When this situation happens (the called function cannot be analysed), we create a summary edge for every call argument (in this case only one summary edge is created, the one for argument `Arg`) and all of them are labelled with an asterisk constraint. This ensures completeness because labelling the summary edges with an asterisk constraint empties the stack when it is traversed, which means that the whole argument is needed.

4.4.3 Dealing with recursion

Another possible source of non-termination is recursion. Direct or indirect recursion produce recursive CE-SDG grammars, which may lead to infinite derivations. This situation can be seen in Figure 4.17.

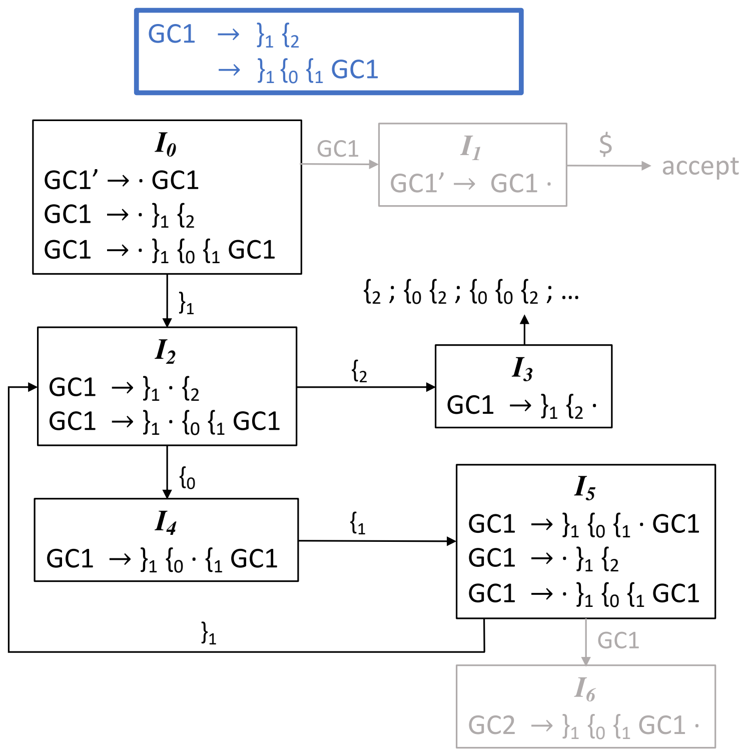


FIGURE 4.17: Infinite derivation in a LR(0) automaton

The grammar in the box at the top corresponds to a grammar constraint `GC1`. This grammar summarizes two paths, being one of them recursive. The LR-(0) analysis produces the LR(0) automaton at the bottom. It has seven nodes, but only those reachable through terminal symbols are relevant (the others are in grey colour). Clearly, we have a loop formed by the sequence $\{_0\{_1\}_1$. Therefore, any input stack that traverses this grammar constraint and can enter infinitely into the loop will produce an infinite sequence of output stacks. For instance, if we traverse this grammar constraint with the stack $\{_1$, then the output stacks produced are infinite (see the output of I_3). The solution to this problem is analogous to the one proposed in the intraprocedural counterpart: dealing with these loops at slicing time with the PDA. The key factor is that, in spite of grammar constraints are defined as a set of productions during summary generation, loops can be detected if the edge is stored together with the constraint in each terminal symbol of the grammar. With this information, increasing loops can be detected also in recursive calls by using the PDA of Figure 4.8. This process is equivalent to the one defined for the CE-PDG, thus, it ensures slicing termination.

4.4.4 Slicing the CE-SDG

The interprocedural CE-SDG produced by Algorithm 4.2 is equipped with new edges and constraints not present in the intraprocedural CE-SDG. These include input, output, and call edges; and also grammar constraints in summary edges.

In this section, we extend Algorithm 4.1 to deal with interprocedural programs. Algorithm 4.3 is based on the slicing algorithm proposed in [85]. As in the original algorithm, there are two different traversal phases and each one ignores a specific kind of edges (Lines 2-3). The behaviour of each slicing phase is very similar to the process described in Algorithm 4.1. The algorithm uses a work list with the states that must be processed, and each state represents the traversal of an edge (node reached, current stack, and list of traversed edges) (Line 9). First of all, function getIncomingEdges (Line 10) collects all the edges that reach the current node. Then, the edges specified by the argument *ignoredEdgeType* (output or input edges, depending on the phase) are ignored as well as the flow edges for which the previous traversed edge was structural (Line 12). After that, the constraint is processed by the processConstraint function (Line 14), which is now augmented to resolve grammar constraints. Grammar constraints summarise all the possible paths from a formal-in to a formal-out, so the traversal of a summary edge with the same initial stack may result into a set of different new stacks (one for each possible path). For this reason, the processConstraint function now returns a list of possible stacks instead of a single stack. All of them are included in the work list to be processed (Line 16) and, finally, the state is added to a list of processed states and the node is added to the work list.

Function processConstraint also detects traversal loops, modifying the value of the stack after processing a particular traversal repeated edge (lines 25-26). After that, all the productions of each grammar constraint are processed with function processProduction, which processes each production constraint by constraint, with a call to function processConstraint, allowing the detection of loops even when processing the elements of grammar constraints in summary edges. Inside processProduction function, function call to getEdge returns the information about the edge corresponding to this constraint, stored in the summary grammar.

Algorithm 4.3 Interprocedural Slicing algorithm for CE-SDGs

Input: A CE-SDG, the slicing criterion node n_{sc}.
Output: The set of nodes that compose the slice.

```
1: function slicingAlgorithmInter(CE-SDG, $n_{sc}$)
2:   $slicePhase1 \leftarrow$ slicingPhase($\{\langle n_{sc}, \bot, null\rangle\}$, $OUTPUT$)
3:   $slicePhase2 \leftarrow$ slicingPhase($slicePhase1$, $INPUT$)
4:   return $\{node \mid \langle node, \_, \_\rangle \in slicePhase2\}$
```

```
function SLICINGPHASE($workList$, $ignoreEdgeType$)
  $processed \leftarrow \emptyset$
  while $workList \neq \emptyset$ do
    select some $state \in workList$
    $\langle node, stack, traversedEdges \rangle \leftarrow state$
    for all $edge \in$ GETINCOMINGEDGES($node$) do
      $\langle sourceNode, type, \_ \rangle \leftarrow edge$
      if $ignoreEdgeType = type \lor$ (GETLASTEDGETYPE($traversedEdges$) $= structural \land type = flow$) then
        continue for all
      $stackSet \leftarrow$ PROCESSCONSTRAINT($stack$, $edge$)
      for all $newStack \in stackSet$ do
        $workList \leftarrow workList \cup \{\langle sourceNode, newStack, traversedEdges$ ++ $edge \rangle\}$
    $processed \leftarrow processed \cup \{state\}$
    $workList \leftarrow workList \setminus processed$
  return $processed$

function PROCESSCONSTRAINT($stack$, $edge$)
  $\langle \_, \_, constraint \rangle \leftarrow edge$
  if $constraint = AsteriskConstraint$ then
    return $\{\perp\}$
  else
    if $edge \in traversedEdges$ then
      if ISINCREASINGLOOP(FINDLOOP(traversedEdges),edge) then return $\{\perp\}$
    if $constraint = EmptyConstraint$ then
      return $\{stack\}$
    else if $constraint = AccessConstraint$ then
      $newStack \leftarrow$ PROCESSACCESS($stack$, $constraint$)
      if $newStack = error$ then
        return $\emptyset$
      else
        return $\{newStack\}$
    else
      $productions \leftarrow$ GETPRODUCTIONS($constraint$)
      return $\bigcup_{p \in productions}$ PROCESSPRODUCTION($stack, p$)

function PROCESSPRODUCTION($stack, production = \langle c_1, \ldots, c_n \rangle$)
  $stacks \leftarrow \{stack\}$
  for all $c_i \in production$ do
    $stacks' \leftarrow \emptyset$
    for all $s \in stacks$ do
      $constraintEdge \leftarrow$ GETEDGE($c_i$)
      $stacks' \leftarrow stacks' \cup$ PROCESSCONSTRAINT($s$, $constraintEdge$)
    $stacks \leftarrow stacks'$
  return $stacks$
```

4.5 Implementation

The changes performed in the PDG/SDG to implement this approach start from a decomposition in the representation of explicit data structures in Section 4.1 (e.g., lists and tuples in Erlang). For this reason, to implement and evaluate this approach, we have used an alternative fine-grained program representation equivalent to the SDG where the elements inside lists and tuples are unfolded into a tree of nodes replicating the structure described in this chapter. This representation is called *Expression Dependence Graph* (EDG) and how the graph is created and how we have implemented a program slicer based on it (e-Knife) is completely described in Chapter 5. Considering this, we have extended the implementation of e-Knife (Chapter 5, Section 5.5) with a new package composed by a set of classes representing different program constraints, included in the edges during the EDG creation. Therefore, we have transformed the EDG into the CE-EDG. Additionally, we have included the implementation of both Algorithms 4.1 and 4.3 in the slicing package.

Figure 4.18 represents the architecture of e-Knife after increasing its representation with constraints and adding the new slicing algorithm. In the figure, bolded squares and bolded rounded squares indicate new classes and packages added to the original e-Knife implementation. The main classes added to the method are the following:

- `ConstrainedAlg`. This class implements Algorithms 4.1 and 4.3, also applicable to the CE-EDG.
- `Constraint`. This set of classes represent the different types of constraints described in Section 4.1. There is one class per constraint and some extra classes to configure the behaviour of each constraint during the slicing process when using the constrained algorithm (Algorithm 4.3).

After the introduced constraint model, the implementation grew from 9600 to a total size of 11000 lines of code approximately. 1 package and a total of 21 new Java classes are added, leaving a project with 72 Java classes distributed in 14 different packages. All the source code is publicly available in `https://mist.dsic.upv.es/git/program-slicing/e-knife-erlang` and a limited online version to test the tool can be found at `https://mist.dsic.upv.es/e-knife-constrained/`.

4.6 Experimental Results

Comparing our implementation against other slicers is not the best way to assess the proposed stack extension to the PDG, because we would find big differences in the PDG construction time, slicing time, and slicing precision due to differences in the libraries used, different treatment for syntax constructs such as list comprehensions, guards, etc. Therefore, we would not be able to assess the specific impact of the stack on the slicer's precision and performance. The only way to do a fair comparison is to implement a single slicer that is able to build and slice the PDG/SDG with and without constraints. This is exactly what we have done.

All the algorithms and ideas described in this chapter have been implemented in a slicer for Erlang called e-Knife. e-Knife can produce slices based on either the PDG/SDG or their CE-counterparts. Thus, it allows us to know exactly the additional cost required to build and traverse the constraints, and

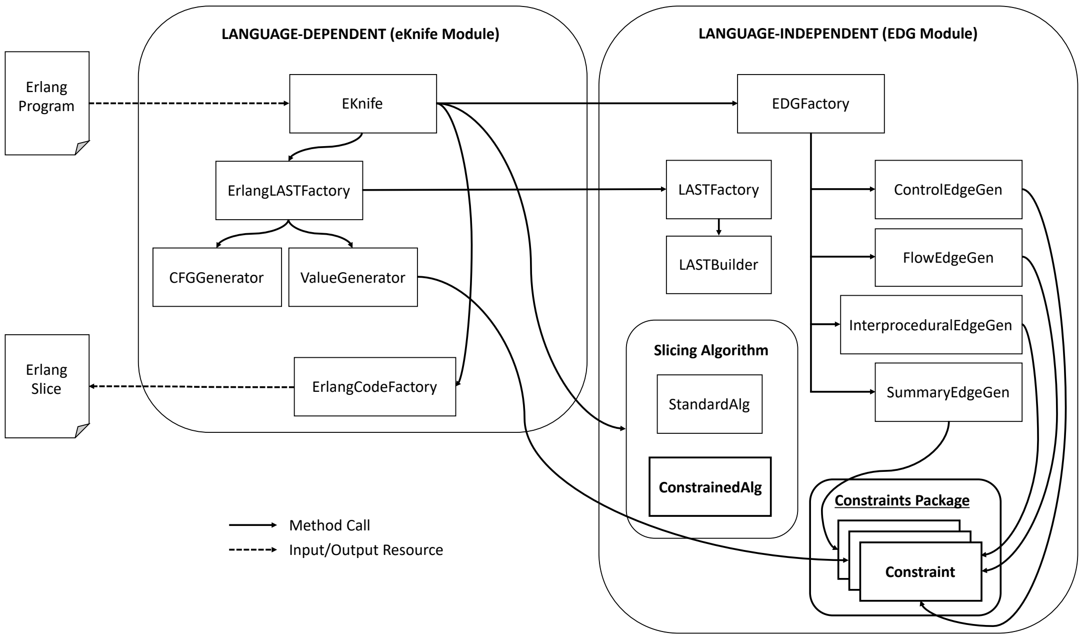


FIGURE 4.18: `e-Knife` architecture after adding constraints and the new slicing algorithms

the extra precision obtained by doing so. e-Knife is a Java program with more than 11000 LOC. It is an open-source project and is publicly available[3].

Additionally, anyone can slice a program via a web interface[4], without the need to build the project locally. Large or very complex programs executed through the web interface may run into memory and time limitations, placed in this interface to avoid abuse.

To evaluate our technique we have divided our experimentation in two parts: intraprocedural evaluation (using Algorithm 4.1), and interprocedural evaluation (using Algorithm 4.3. To test the performance of e-Knife, we used Bencher, a program slicing benchmark suite for Erlang. All the benchmarks were interprocedural programs, so, to conduct the intraprocedural evaluation, we have created a new intraprocedural version of them (by inlining functions). This intraprocedural version has been made publicly available[5]. To evaluate the techniques proposed throughout this work, we have built both graphs (PDG/SDG and CE-PDG/CE-SDG) for each of the intra and interprocedural programs in bencher. Then, we sliced both graphs with respect to all possible slicing criteria[6], which guarantees that there is no bias in the selection of slicing criteria.

We strictly followed the methodology proposed by Georges et al. [67]. Each program's graph was built 1001 times, and the graphs were sliced 1001 times per criterion. To ensure real independence, the first iteration was always discarded (to avoid influence of dynamically loading libraries to physical memory, data persisting in the disk cache, etc.). From the 1000 remaining iterations we retained a window of 10 measurements when steady-state performance was

[3]`https://mist.dsic.upv.es/git/program-slicing/e-knife-erlang`
[4]`https://mist.dsic.upv.es/e-knife-constrained/`
[5]`https://mist.dsic.upv.es/bencher/`
[6]Each variable use or definition in all functions that contain complex data structures.

	Graph Generation		Slice					
Program	**PDG**	**CE-PDG**	**Function**	**#SCs**	**PDG**	**CE-PDG**	**Slowdown**	**Red. Size**
`bench1A.erl`	5468.10ms	5474.38ms	`getLast/2`	26	82.55μs	392.39μs	4.40 ± 0.50	$14.88 \pm 3.23\%$
			`getNext/3`	174	308.66μs	1645.44μs	4.94 ± 0.16	$13.06 \pm 1.58\%$
			`getStringDate/1`	11	30.18μs	93.71μs	3.22 ± 0.18	$8.67 \pm 4.07\%$
			`main/1`	57	1121.93μs	2869.79μs	2.59 ± 0.30	$38.76 \pm 7.22\%$
`bench3A.erl`	49.58ms	49.59ms	`tuples/2`	22	38.39μs	153.21μs	3.69 ± 0.44	$5.46 \pm 2.08\%$
`bench4A.erl`	79.70ms	79.76ms	`main/2`	31	89.80μs	376.33μs	4.23 ± 0.42	$20.79 \pm 5.47\%$
`bench5A.erl`	48.69ms	48.73ms	`lists/2`	18	60.92μs	265.87μs	3.82 ± 0.43	$6.51 \pm 2.08\%$
`bench6A.erl`	403.52ms	403.66ms	`ft/2`	34	82.59μs	333.51μs	3.60 ± 0.36	$12.25 \pm 2.72\%$
			`ht/2`	16	21.39μs	71.55μs	2.94 ± 0.29	$10.79 \pm 3.81\%$
`bench9A.erl`	199.53ms	199.71ms	`main/2`	18	197.94μs	458.68μs	2.25 ± 0.14	$1.38 \pm 1.07\%$
`bench11A.erl`	15.49ms	15.52ms	`lists/2`	16	43.09μs	141.87μs	3.30 ± 0.16	$6.47 \pm 2.22\%$
`bench12A.erl`	1661.91ms	1663.25ms	`add/4`	26	104.88μs	454.55μs	4.27 ± 0.49	$15.21 \pm 4.29\%$
			`from_ternary/2`	9	22.92μs	103.17μs	4.28 ± 0.44	$3.56 \pm 2.76\%$
			`main/3`	39	103.42μs	408.30μs	4.03 ± 0.51	$8.43 \pm 6.27\%$
			`mul/3`	21	55.05μs	261.14μs	4.57 ± 0.35	$2.74 \pm 1.31\%$
			`to_ternary/2`	13	71.93μs	199.70μs	3.05 ± 0.28	$1.02 \pm 1.37\%$
`bench14A.erl`	3841.95ms	3842.62ms	`main/2`	81	85.94μs	451.66μs	4.01 ± 0.40	$8.76 \pm 2.56\%$
`bench15A.erl`	1948.76ms	1949.37ms	`main/4`	71	246.97μs	609.24μs	2.94 ± 0.19	$2.31 \pm 1.73\%$
`bench16A.erl`	276.60ms	276.79ms	`word_count/5`	36	83.79μs	289.83μs	3.96 ± 0.30	$8.91 \pm 2.93\%$
`bench17A.erl`	63.47ms	63.60ms	`mug/3`	19	55.44μs	202.33μs	3.78 ± 0.18	$5.59 \pm 3.10\%$
`bench18A.erl`	71.38ms	71.50ms	`mbe/2`	19	83.69μs	278.30μs	3.73 ± 0.31	$7.38 \pm 4.71\%$
			Totals and averages for set A	**757**	218.65μs	814.51μs	3.88 ± 0.10	**11.67±3.02%**

TABLE 4.3: Summary of experimental results, comparing the PDG (without constraints) to the CE-PDG (with constraints).

reached, i.e., once the coefficient of variation (CoV, the standard deviation divided by the mean) of the 10 iterations falls below a preset threshold of 0.01 or the lowest CoV if no window reached it. It is with these 10 iterations that we computed the average time taken by each operation (building each graph or slicing each graph w.r.t. each criterion).

The results of the experiments performed are summarised in Tables 4.3 and 4.4. These tables sum up two experiment sets, one for the intraprocedural suite (called set A) and other for the interprocedural suite (called set B). The two columns into the graph generation part of the tables display the average time required to build each graph. Building the CE-PDG/CE-SDG, as in the PDG/SDG, is a quadratic operation; and the inclusion of labels in the edges is a linear operation. Thus, building the constrained graphs is only slightly slower than its counterpart. The other columns are as follows (average values are w.r.t. all slicing criteria):

Function (only in Table 4.3): the name of the function where the slicing criterion is located.

#SCs: the number of slicing criteria in that function.

PDG, CE-PDG, SDG, CE-SDG: the average time required to slice the corresponding graph.

Slowdown: the average additional time required (with 95% error margins), when comparing the CE-PDG/CE-SDG with the corresponding PDG/SDG. For example, in the first row of Table 4.3, the computation of each slice is on average 4.40 times slower in the CE-PDG.

Red. Size: the average reduction in the slices sizes (with 95% error margins). It is computed as $(A - B)/A$ where A is the size (number of AST nodes) of the slice computed with the standard (field-insensitive) algorithm and

	Graph Generation		**Slice**				
Program	**SDG**	**CE-SDG**	**#SCs**	**SDG**	**CE-SDG**	**Slowdown**	**Red. Size**
`bench1B.erl`	4689.59ms	4695.39ms	273	2375.91μs	52978.07μs	19.04 ± 1.48	$8.47 \pm 2.33\%$
`bench2B.erl`	122.07ms	122.10ms	17	100.30μs	160.02μs	2.54 ± 0.47	$0.62 \pm 0.65\%$
`bench3B.erl`	53.70ms	53.71ms	18	73.09μs	283.20μs	3.70 ± 0.42	$6.15 \pm 3.65\%$
`bench4B.erl`	38.34ms	38.40ms	39	136.43μs	351.29μs	2.98 ± 0.33	$12.03 \pm 3.61\%$
`bench5B.erl`	24.67ms	24.72ms	11	83.64μs	316.45μs	3.83 ± 0.20	$6.88 \pm 0.85\%$
`bench6B.erl`	89.36ms	89.49ms	44	64.04μs	241.37μs	3.65 ± 0.39	$7.67 \pm 2.07\%$
`bench8B.erl`	144.54ms	144.67ms	42	317.21μs	19641.19μs	57.75 ± 7.30	$0.73 \pm 0.68\%$
`bench9B.erl`	53.57ms	53.65ms	17	305.20μs	588.48μs	2.02 ± 0.16	$1.38 \pm 1.04\%$
`bench10B.erl`	146.72ms	146.98ms	35	415.38μs	7368.92μs	26.06 ± 5.94	$2.48 \pm 1.25\%$
`bench11B.erl`	15.10ms	15.15ms	13	69.71μs	248.10μs	3.58 ± 0.18	$8.02 \pm 2.08\%$
`bench12B.erl`	526.36ms	527.29ms	88	1445.05μs	7244.07μs	5.15 ± 1.32	$5.10 \pm 2.85\%$
`bench13B.erl`	41.00ms	41.05ms	22	212.20μs	307.64μs	1.88 ± 0.35	$8.36 \pm 7.31\%$
`bench14B.erl`	257.98ms	258.50ms	52	167.99μs	522.23μs	3.20 ± 0.40	$13.58 \pm 4.50\%$
`bench15B.erl`	376.22ms	376.62ms	73	394.71μs	770.11μs	2.39 ± 0.16	$10.08 \pm 2.88\%$
`bench16B.erl`	170.25ms	170.42ms	40	200.22μs	3490.60μs	30.73 ± 6.76	$3.70 \pm 1.51\%$
`bench17B.erl`	93.42ms	93.55ms	19	248.47μs	442.49μs	1.88 ± 0.22	$4.96 \pm 2.38\%$
`bench18B.erl`	102.34ms	102.48ms	19	393.15μs	607.97μs	1.55 ± 0.15	$0.05 \pm 0.10\%$
Totals and averages for set B			**816**	1060.16μs	19742.28μs	13.43 ± 1.18	**7.12±2.47%**

TABLE 4.4: Summary of experimental results, comparing the SDG (without constraints) to the CE-SDG (with constraints).

B is the size (number of AST nodes) of the slice computed with the field-sensitive algorithm (Algorithms 4.1 and 4.3 for both Tables 4.3 and 4.4, respectively). This way of measuring the size of the slices is much more precise and fair. LOC is not proper because it can ignore the removal of subexpressions. PDG/CE-PDG (respectively SDG/CE-SDG) nodes is not a good solution because the CE-PDG includes nodes and arcs not present in their counterparts, therefore they are incomparable.

The averages shown at the bottom of the table are the averages of all slicing criteria, and not the averages of each function's average.

The first 13 benchmarks (set A) are benchmarks with complex data structures sliced using the intraprocedural algorithm, while the rest of benchmarks (set B) are programs with multiple functions connected by interprocedural edges. In set A, each slice produced by the CE-PDG is around four times slower. However, this has little impact, as each slice consumes just hundreds of milliseconds. As can be seen in each row, generating the graph is at least 3 orders of magnitude slower than slicing it. This increase in time is offset by the average reduction of the slices, which is $11.67 \pm 3.02\%$ at almost no cost (only a few μs). This increase goes up to 38.76% in function `main/1` from `bencher1A`, as it contains complex data structures that can be efficiently sliced with the CE-PDG. The same happens in set B, but due to the overhead included when resolving grammar constraints, the slowdown is around thirteen times slower.

If we consider interprocedural program slicing, the slowdown is 13.43. In this case, the technique has more opportunities for improvement because, contrarily to the intraprocedural CE-PDG, the graph includes the traversal of summary edges, where it is more common to run a cycle detection process due to the appearance of summary grammars. Even so, the use of constraints results into a reduction in the slices size of $7.12 \pm 2.47\%$.

If we consider both sets, the average reduction of using the CE-PDG/CE-SDG and their corresponding algorithms results in an average reduction of

9.31±2.73%. This is a very good result: for many applications, e.g., debugging, reducing the suspicious code over 9.31% with a cost of increasing the slicing time by only a few milliseconds is a good trade-off to make.

4.7 Related Work

Transitive data dependence analysis has been extensively studied [167, 187]. Less attention has received, however, the problem of field-sensitive data dependence analysis [108, 123, 163, 186]. The existing approaches can be classified into two groups: those that treat composite structures as a whole [123, 129, 139, 147], and those that decompose them into small atomic data types [2, 9, 18, 72, 100, 103, 138, 163]. The later approach is often called *atomization* or *scalar replacement*, and it basically consists of a program transformation that recursively disassembles composite structures to their primitive components. However, slicing over the decomposed structures usually uses traditional dependence graph based traversal [9, 72, 103] which limits the accuracy. Other important approaches for field-sensitive data dependence analysis of this kind are [108, 123, 186]. Litvak et al. [123] proposed a field-sensitive program dependence analysis that identifies dependences by computing the memory ranges written/read by definitions/uses. Späth et al. [186] proposed the use of pushdown systems to encode and solve field accesses and uses. Snelting et al. [183] present an approach to identify constraints over paths in dependence graphs. Our approach combines atomization with the addition of constraints checked by pushdown systems to improve the accuracy of slicing composite data strauctures.

There exist approaches for the field-sensitive slicing of some specific data structures. If we refer to arrays, some static proposals consider the whole array as a variable, and each access as a definition or use of that variable [129]. However, this technique produces complete, but unnecessarily large program slices [18]. The PDG variant of Ottenstein and Ottenstein [147] represents composite data types providing a node for each one of its subexpressions, and provides special *select* and *update* operators to access the elements of an array. Other static approaches rely on determining whether two statically unknown vector accesses can refer to the same memory location during runtime [39, 111]. Some papers [14, 133, 158] propose algorithms that demonstrate the absence of a flow dependence between array accesses under certain conditions.

Some approaches [33, 37, 72, 112] have been also proposed to accurately represent the inner structure of objects and the dependences between their data members. Most object-oriented approaches [72, 118, 202] are based on the same principle: object variables and their inner data members are unfolded in a tree-like representation when used at function calls. This allows for the generation of dependences between data members of a particular object and to accurately slice off those data members of an object that are not affecting the slicing criterion. Our representation is inspired by this tree-like structure, but with some differences. In our representation the tree structure is connected with a new kind of edges (structural edges) instead of control edges. This allows us to apply a different slicing behaviour for structural edges without interfering in the traversal restrictions given to control edges in some slicing algorithms [109].

Additionally, our tree structure is connected not only with structural edges, but also with flow edges; providing a more realistic representation of the dependences between a composite structure and all its elements.

Severals works have tried to adapt the PDG for functional languages dealing with tuple structures in the process [25, 32, 108, 198]. Some of them with a high abstraction level [171], and other ones with a low granularity level. Silva et al. [180] propose a new graph representation for the sequential part of Erlang called the Erlang Dependence Graph. Their graph, despite being built with the minimum possible granularity (each node in the graph corresponds to an AST node) and being able to select subelements of a given composite data structure, does not have a mechanism to preserve the dependence of the tuple elements when a tuple is collapsed into a variable; i.e., they do not solve the *slicing pattern matching* problem (for instance, they cannot solve the program in Figure 4.2). In contrast, although our graph is only fine-grained at composite data structures, we overcome their limitations by introducing an additional component to the graph, the constrained edges, which allow us to carry the dependence information between definition and use even if the composite structure is collapsed in the process.

Chapter 5

Overcoming SDG Limits: The Expression Dependence Graph

Besides completeness (extracting all the code that affects or is affected by the slicing criterion), the main goal of program slicing is precision (removing as much code that does not affect or is not affected by the slicing criterion as possible). Nevertheless, most implementations of program slicing use the SDG, which imposes a precision barrier: its granularity level are statements. Currently, the only statements that are broken down properly in the SDG are objects, blocks, and procedure calls. By breaking down procedure calls, program slicing can reason about whether or not the arguments of a specific call should be sliced off. However, other important situations cannot be handled by the SDG, and some of them cannot be even correctly handled by industrial refactoring tools that include program slicing. In this chapter we showcase some of these situations (Section 5.1) and present a new representation model, the Expression Dependence Graph (EDG) to solve them. We describe how the EDG is created and sliced in Sections 5.2 and 5.3. After that, in Section 5.4 we illustrate how the EDG representation solves the problems presented in Section 5.1. Finally, we show the implementation and the results of the experimental evaluation of the model in Sections 5.5 and 5.6.

5.1 Representation problems of the SDG

Along this section we show that the SDG is implicitly limited by its way of representing complex syntax constructs. We present some problems of the SDG when representing code of both imperative and functional programs with examples in Java and Erlang. In general, the problems shown apply to any language that has similar syntax constructs. For each problem, we compare the slice obtained with the SDG and the slice computed with our graph representation. In each fragment of code the underlined blue code represents the slicing criterion, the computed slice is shown in black, and the sliced code is shown in grey.

Problem 5.1 (Extraction of embedded definitions)**.** *Figure 5.1 contains a fragment of code extracted from [55]. This code shows the SDG's lack of accuracy in assignments with several operands. As the SDG represents the whole assignment with a single node, the definition of variable b in line 4 cannot be sliced without including the whole d assignment statement. Note that including the procedure*

calls `foo()` and `baz()` in the slice is very imprecise because it triggers a snowball effect, forcing the slice to also include the definition of these functions and all other functions called from them.
Cause: The SDG represents assignments with multiple definitions as a single node.
Current solution: A solution to this problem is shown in [97], where the authors define a new graph representation called the combined C graph, *where embedded definitions with side effects are moved to another graph node to improve slicing precision. An alternative solution for this problem is to perform an ad-hoc program transformation: line 4 can be replaced by: `d = a * b * c; b++;`. This transformation, however, can be insufficient in many cases such as: `d = a * b++ * c + b;` and, thus, further analysis is needed.*

```
1  a = foo();
2  b = bar();
3  c = baz();
4  d = a * b++ * c;
5  e = b;
```

(A) SDG slice

```
1  a = foo();
2  b = bar();
3  c = baz();
4  d = a * b++ * c;
5  e = b;
```

(B) EDG slice

FIGURE 5.1: Extract inner assignments

Problem 5.2 (Monolithic code). *Consider the simple Java source code fragment of Figure 5.2. Selecting a variable in an expression as the slicing criterion may force the inclusion in the slice of some variables and expressions that are not actually needed. In this case, because the condition contains a non-short-circuit operand, the evaluations of neither `x = 0`, `x = 10`, `x == 10` nor `foo()` are actually needed to reach variable `y` in line 3. Therefore, in this example it is not enough to decompose the if-condition into the two boolean subexpressions. To make the slice precise, the extraction of a single variable is needed.*
Cause: The SDG does not decompose expressions, thus it prevents from analysing the inter-dependences between operations and operands.
Current solution: This lack of precision is often assumed even by commercial slicers. A solution is to post-process the slice code ad-hoc to determine what operands could be removed. This often implies a data flow analysis that not always ensures an improvement in the accuracy.

```
1  int x = 0, y = 5;
2  x = 10;
3  if (x == 10 | y == foo()) {
4     x = y = 20;
5  }
```

(A) SDG slice

```
1  int x = 0, y = 5;
2  x = 10;
3  if (x == 10 | y == foo()) {
4     x = y = 20;
5  }
```

(B) EDG slice

FIGURE 5.2: Unnecessary evaluation

Problem 5.3 (`try-catch` structures). *Figure 5.3 shows an important problem that makes the SDG to produce wrong (incomplete) slices. The only possible*

executions that reach the slicing criterion are those in which `f(a)` *produces an exception. However, if* `f(a)` *produces an exception, then* `b` *is never assigned in line 4. This means that the value of* `b` *in the slicing criterion (a use) comes from the definition of* `b` *in line 2 (this is correctly sliced in the EDG slice). Unfortunately, in the SDG, the assignment in line 4 is represented with a node that includes* `b` *and the call to* `f(a)`*. Thus, both are included in the slice, being the definition of* `b` *in line 4 not only unnecessary, but a source of incompleteness.*
Cause: The SDG node raising the exception and the SDG node defining `b` *are the same. For this reason the last definition of* `b` *is always found in this node whether the exception is raised or not. Thus, this definition of* `b` *prevents its previous definition to be considered.*
Current solution: A solution to this problem was proposed by Allen and Horwitz [6]. The solution augments the CFG of method calls with two different paths for smooth and exception executions. This path division moves the definition of `b` *in line 4 exclusively to a path where the exception is not raised. This way, when the exception is raised, the last life definition for the use of* `b` *in the* `catch` *block, is the one in line 2. Another valid approach would be to use a pre- and post-processing of the SDG that analyse what parts of the code are not executed when exceptions are thrown. This analysis should be used at slicing time to avoid including unfinished assignments, like the one for* `b` *in line 4, in the slice.*

```
1  int b;
2  b = 10;
3  try {
4     b = f(a);
5  } catch (Exception e) {
6     log(b);
7  }
```

(A) SDG slice

```
1  int b;
2  b = 10;
3  try {
4     b = f(a);
5  } catch (Exception e) {
6     log(b);
7  }
```

(B) EDG slice

FIGURE 5.3: `try-catch` structures

Problem 5.4 (`for` loops)**.** *Figure 5.4 shows that the SDG by default includes in the slice the three components of the* `for`*-loop's header (initialisation, condition, and update). If one of the components contains more than one element then all of them are included in the slice.*
Cause: The SDG represents complex program constructs with a single node.
Current solution: A solution was proposed in [132], where a specific representation in the SDG for the `for` *loop is presented. Roughly, they decompose with independent nodes the initialisation, the condition, and the update (this is in the same direction as the EDG) but they fail to further decompose the expressions inside them.*

Problem 5.5 (List comprehensions)**.** *Figure 5.5 shows that the SDG lacks of a representation for many program constructs such as the list comprehension. In the standard SDG, the whole assignment, including the list comprehension is part of a single node, which leads to a big source of imprecision (see Figure 5.5a).*

```
1  it = 0;
2  for (x = 1; x < 10; x += 2, it++) {
3     print(x);
4  }
5  print(it);
```

(A) SDG slice

```
1  it = 0;
2  for(x = 1; x < 10; x += 2, it++) {
3     print(x);
4  }
5  print(it);
```

(B) EDG slice

FIGURE 5.4: `for` loop

Some advanced slicers, especially those designed for functional languages such as SlicErl [181], give an ad-hoc treatment to program comprehension constructs. But even this special treatment is not enough in many cases such as this. Figure 5.5b shows the slice produced by SlicErl, which represents every generator with a single node, thus wrongly including `Y` *in the slice.*
Cause: The SDG does not explicitly represent list comprehensions because they are expressions.
Current solution: Ad-hoc representations in the SDG for different constructs such as list comprehensions, generators, filters, or just tuples are used in the Erlang functional language in Silva et al. [181].

```
[[X,Y] || {X,Y} <- L1, X < 1, Y > 2]
```

(A) SDG slice

```
[[X,Y] || {X,Y} <- L1, X < 1, Y > 2]
```

(B) SlicErl slice

```
[[X,Y] || {X,Y} <- L1, X < 1, Y > 2]
```

(C) EDG slice

FIGURE 5.5: List comprehension

This set of problems show important limitations of the SDG, being the worst that the minimum granularity level is a statement. The SDG does not allow us to represent operations, literals, variables, or expressions; nor to select them as the slicing criterion; nor to remove them from the slice when they are part of a node that belongs to the slice.

We have observed that similar ad-hoc solutions have been proposed in the literature once and again to solve the same problem in different contexts. To cite some examples, the SDG has been extended with additional nodes to represent: objects [118, 128, 202], exceptions [6, 62], `for`-loops [132], list comprehensions [181], composite data structures [72, 108, 123, 186], etc. The need for these ad-hoc solutions are a clear symptom that the SDG should be generalised to account for all those problems. We propose a generalisation of the graph that not only allows us to solve all those problems at once, but also enables the graph to reason about problems for which the SDG is not suitable, like the representation of isolated variable declarations as an independent node of the graph.

Our extension of the SDG is a natural generalisation that makes it more precise because each single literal (expression) is represented with a node (i.e., each node represents the minimum programmatic information: an abstract syntax tree (AST) node), for this reason we call this extension the *expression dependence graph* (EDG). This new representation naturally produces a solution to most of the above problems. E.g., `for`-loops and list comprehensions are trivially decomposed and solved; complex data structures such as lists or tuples are broken down allowing us to distinguish different elements inside them; etc.

The EDG is a fine-grained representation that incorporates several novelties:

- Each EDG node represents the minimum program unit: a program literal.
- The EDG defines a new type of dependence (called *value dependence*) that only exists between subexpressions.
- In the EDG, expressions and statements are represented in a different way.
    - Expressions have a dual-node representation where one node contains the value given to the literal it represents, and the other the presence of the literal in the program. This representation provides more possibilities to naturally obtain executable well-structured slices including compilation dependences.
    - Statements have a single-node structure that represents their presence or absence in the slice.
- The EDG includes another new type of dependence (called *declaration dependence*) that is able to include variable declarations (without including the initial value if it is not needed) taking profit of the dual-node representation given to expressions.
- The EDG provides an explicit representation for function returned values, naturally building "result" nodes for every function definition and function call.

Because the proposed extension is an SDG generalisation, the new representation can behave exactly as a standard SDG (the granularity of the nodes can be reduced by ignoring some edges, or by treating some subtrees as a single node). Therefore, all the extensions defined so far for the SDG are still valid in the EDG. However, they could be reformulated taking advantage of the new EDG nodes to improve their accuracy.

It is important to remark that we do not propose here a developed solution to all those slicing problems; but a tool—a new program representation—that can be used to better solve many slicing problems. For this reason, in order to evaluate the initial potential of our representation, we have focused our implementation and evaluation in a subset of two programming languages from different programming paradigms: Java, in representation of the object-oriented paradigm; and Erlang, in representation of the functional paradigm. It is worth mentioning that it is not an objective of this work to implement a slicer for the whole Java and Erlang languages, but only to evaluate the advantages of our representation in different programming paradigms. Hence, our implementation nor considers multiple-class Java programs, neither concurrent Erlang programs; but only considers the imperative subset of Java and the sequential part of Erlang programs (as it happens in the problems shown so far in this section). The implementation of all object-oriented and concurrent programs based on the EDG would be an interesting future work because the performance and the accuracy provided by the EDG in the experimental evaluation has been proved promising.

5.2 From ASTs to EDGs

In this section, we explain how to obtain an EDG from any given AST. Briefly explained, firstly, the AST is labelled with control flow and a new intra-expression dependence called value dependence to produce a labelled AST (L-AST). Secondly, the L-AST is augmented with a dual-node representation for expressions, enhancing the graph expressivity and adjusting the graph for control and flow dependence generation, producing a transformed labelled AST (TL-AST). Thirdly, control, flow, and some extra slicing edges are added to generate an intraprocedural EDG and, finally, as in the SDG, the EDG is augmented with call, parameter-in, parameter-out, and summary edges to be an interprocedural EDG. In a nutshell:

$$\begin{array}{ccccccccc} AST & \rightarrow & L\text{-}AST & \rightarrow & TL\text{-}AST & \rightarrow & EDG & \rightarrow & EDG \\ & & & & & & \text{(intra)} & & \text{(inter)} \end{array}$$

In the following sections, we explain the information a L-AST should contain and how to transform an L-AST into an interprocedural EDG.

The starting graph of the EDG: labelled ASTs

An AST is a tree representation of a source code written in one specific language but, unfortunately, the information provided by its edges (called structural edges from here on) in not enough to compute program slices. For instance, it lacks information about control flow, which is needed to construct control and flow dependences. Therefore, we need the given AST to be equipped with two resources:

1. The control flow graph (CFG). The CFG is the starting graph needed to construct the whole PDG of a method; whereas, in the EDG, the CFG is an additional resource incorporated into the AST indicating in which order AST nodes are evaluated during the execution. As it happens with the PDG, the CFG is completely necessary to compute control and flow dependences.

2. The value edges that represent what we call *value dependence*. Value dependences are new intra-statement dependences that appear when breaking down some PDG nodes. They were already noted by Krinke and Snelting in [103, 107]. Value dependences indicate where the value of an expression comes from through its components. They are language-dependent, represent how the values flow between the components of an expression, and are essential to keep slices precise when it comes to subexpressions.

First of all, we need to compute the CFG for each procedure of the program, defining the order in which the AST nodes are executed. The nodes of the CFG used in the EDG are AST nodes (instead of statements) and the definition of this CFG is formalised below.

Definition 5.1 (Control-flow Graph in ASTs). *Let $G = (N, E_s)$ be the AST of a procedure where N is the set of nodes of the procedure and E_s is the set of structural edges in the AST of the procedure. Let n_1 and n_2 be nodes in N. The CFG of G is a graph $CFG_G = (N, E_{cf})$ where the set E_{cf} fulfils the following conditions:*

1. $(n_1, n_2) \in E_{cf} \iff n_2$ *can be immediately executed after* n_1
2. $\exists n_{enter} \in N \mid \forall n \in N \ . \ (n_{enter}, n) \in E^*_{cf}$
   {All procedure nodes can be reached from an initial node Enter by traversing control-flow edges}
3. $\exists n_{exit} \in N \mid \forall n \in N \ . \ (n, n_{exit}) \in E^*_{cf}$
   {The Exit node can be reached from any procedure node by traversing control-flow edges}
4. $n_{enter} = n_{exit} = root(G)$
   {The enter *and the* exit *node must be the same node, the root of the AST}*

We can now define the new dependence (value dependence) that appears in a CFG defined over an AST. Given an AST G, we define the set *EXP* as the set of nodes in G that represent expressions.

Definition 5.2 (Value Dependence). *Let $G = (N, E_s \cup E_{cf})$ be an AST with an inscribed CFG where N is a set of nodes, E_s is a set of structural edges, and E_{cf} a set of control-flow edges. Given two expression nodes $n_1, n_2 \in EXP$, n_2 is* value dependent *on a CFG node n_1 if and only if:*

1. *the value of the expression at n_1 is needed to compute the value of n_2, and*
2. $\exists n_3 \in EXP$ *where* $(n_3, n_1), (n_3, n_2) \in E^*_s$*, and*
3. $\nexists n_4 \in N, n_4 \notin EXP$ *where*
   - $(n_3, n_4), (n_4, n_1) \in E^*_s$*, or*
   - $(n_3, n_4), (n_4, n_2) \in E^*_s$

Note that value dependence is transitive and it is restricted to expressions. Condition 2 forces that n_1 is a subexpression of n_2 (when $n_3 = n_2$), or n_2 is a subexpression of n_1 (when $n_3 = n_1$), or n_1 and n_2 are subexpressions of a common expression n_3 (when $n_1 \neq n_2 \neq n_3$). Therefore, value dependence is limited to connect elements within the same statement. Moreover, condition 3 ensures that n_1 and n_2 share a parent expression n_3 without any intermediate statement n_4 between them.

In order to represent value dependence in the AST, we label it with *value edges*, which account for value dependence in an intransitive way, exactly as flow edges do (i.e., two edges `A` $\rightarrow$ `B` and `B` $\rightarrow$ `C` avoid the creation of `A` $\rightarrow$ `C`, thus, minimizing the number of edges). Value edges are the transitive reduction of the value dependence, and they can be computed according to the following definition:

Definition 5.3 (Value edge)**.** *Let $G = (N, E)$ be an AST where N is a set of nodes and E is a set of edges. There is a* value edge *from a node $n_2 \in N$ to a node $n_1 \in N$ if and only if:*

1. *n_1 is value dependent on n_2,*

2. *$\nexists n_3 \in N$. n_1 is value dependent on n_3 and n_3 is value dependent on n_2.*

Value dependence is intra-statement and it complements flow-dependence, which is inter-statement. Both play a similar role, while flow dependence links definitions and uses of variables between different program statements, value dependence works at expression level, linking the subexpressions that are interdependent. Together, they form the so-called data dependence. Observe that value dependence is defined in a language agnostic manner thanks to condition (1). The semantics of each language defines what nodes of an expression are needed to compute the value of the other nodes. Example 5.1 contains an illustrative example of what value dependence represents and how value edges and flow edges represent dependences with different origins.

Example 5.1. *Consider the code in Figure 5.6 that shows (a) two consecutive statements of a program and (b) its associated AST, which incorporates the corresponding value and flow edges.*

```
   ...
x = 1;
y = x + 3;
   ...
```

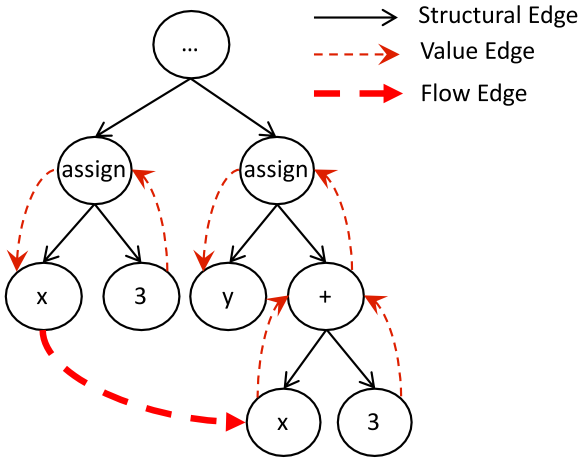


FIGURE 5.6: Two consecutive statements of a program and AST with value and flow dependences

As it can be seen, there is a value edge between the components of each assignment: (`1` --→ `=`)*,* (`=` --→ `x`)*,* (`+` --→ `=`)*,* (`=` --→ `y`)*,* (`x` --→ `+`)*, and* (`3` --→ `+`)*. But not between the* `x` *components of the two different assignments, which are connected by a flow edge.*

```
X = if (Y>0) -> 1;
       true -> 2
end.
```

FIGURE 5.7: Example of if-then-else structure in Erlang

The definition of value dependence can be adapted to each AST considering the collection of expressions used in each language, thus, each language defines its own value dependence. For instance, in Erlang, contrarily to Java, an **if-then-else** is an expression. Therefore, in Erlang, the expression in Figure 5.7 induces a value dependence between `X` and the **if-then-else**.

Another interesting example is the multiple assignment in line 4 of Figure 5.2. Value edges represent how variables `x` and `y` get their values (e.g., 20 is assigned to `y` and later assigned to `x`) during the execution of the assignment. Example 5.2 shows four possible ASTs that represent this assignment. The example shows how each ASTs can be later labelled with control flow and value edges in multiple ways, differentiating which ones provide a representation usable to reach our goal.

Example 5.2 (Possible labelled ASTs for line 4 in Figure 5.2). *Consider Figure 5.8 that contains four possible ASTs for statement `x = y = 20;`.*

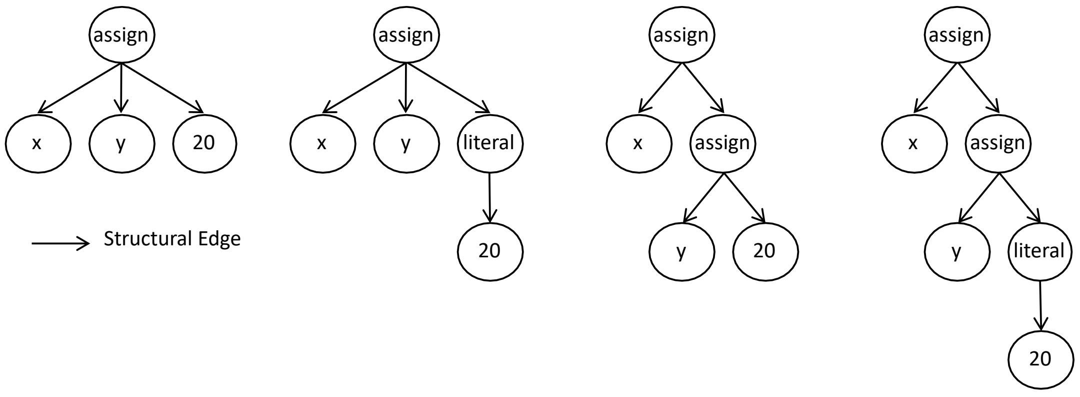


FIGURE 5.8: ASTs for statement `x = y = 20;`

Note that the ASTs in Figure 5.8 are just possible instances, and that there can be other ASTs that represent the same statement. ASTs shown in Figure 5.8 allow for having a representation of the source code, but they do not provide any information about how the source code is executed or how variables get values. Figure 5.9 shows the same ASTs labelled with some extra edges that provide this information. These labelled ASTs are called L-ASTs.

Each AST shown in Figure 5.9 can be labelled in multiple ways. Unfortunately, we cannot discuss all of them because of the exponential combination. Therefore, only some common L-ASTs are shown. In all the L-ASTs, the control flow edges indicate that the evaluation of `20` comes first, and the assignments of `x` and `y` happen afterwards as well as the order in which they are assigned. Value edges indicate where the value of an expression comes from and they are language dependent. For instance, given a specific language, it is needed to define how `if ((x = y = 20) > z)` should be evaluated to finally evaluate `if`

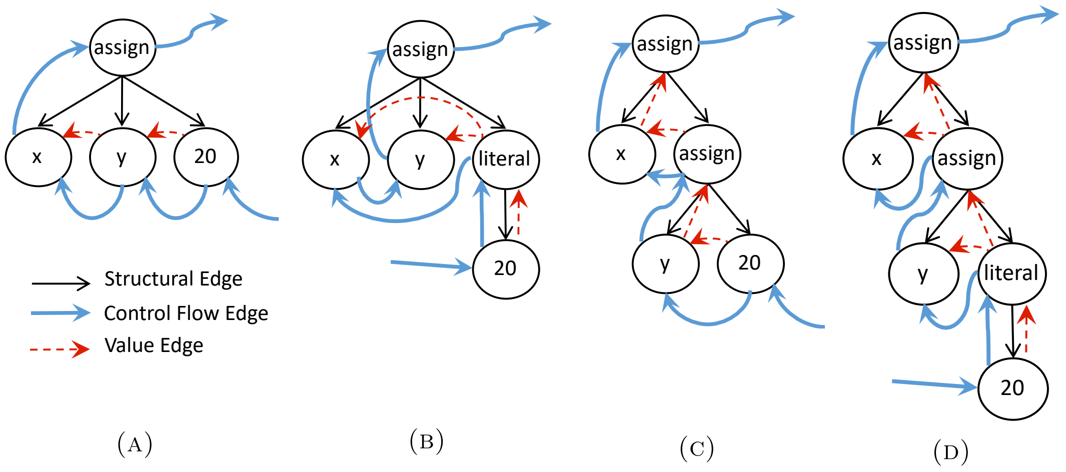


FIGURE 5.9: ASTs of Figure 5.8 labelled with control flow and value edges

(20 > z). In the four ASTs in Figure 5.9, only 5.9c and 5.9d can be used to represent the previous `if` *predicate because they provide a value for the* `(x = y = 20)` *expression. In Figure 5.9c, the value edges indicate that 20 is assigned to* `y`*, the value of* `y` *is later assigned to* `x` *and finally the value of* `x` *would be used in the comparison. In Figure 5.9d, 20 is assigned to* `y`*, assigned to* `x` *and would be used in the comparison, but the order in which this happens is not even imposed (the order can be deduced from the control flow edges).*

To construct an EDG, we start from an AST labelled with control flow and value edges. Control flow edges need to fulfil one important property to correctly calculate control dependences and, thus, control edges: all nodes in a labelled AST must be reachable from an initial node (in Figure 5.9 the initial node is 20). Labelled ASTs are formalised in Definition 5.4.

Definition 5.4 (Labelled AST). *A labelled AST (L-AST) is a graph $G = (N, E)$ where N is a set of nodes in the AST and $E = E_s \cup E_{cf} \cup E_v$ is a set of edges, in which E_s are structural edges, E_{cf} are the control-flow edges computed following the rules in Definition 5.1 for all procedures, and E_v are value edges computed following Definition 5.3.*

Moreover, some nodes in the L-AST have to be labelled indicating what kind of nodes they are. These labels are used to generate the so-called *DEC*, *DEF*, *USE*, and *EXP* sets that contain variable-declaration nodes, variable-definition nodes, variable-use nodes, and expression nodes, respectively. For instance, in Figure 5.9, all nodes belong to the *EXP* set, and the `x` and `y` nodes also belong to the *DEF* set. Any AST labelled with these node sets and with control flow and value edges is valid to be handled by our technique.

Transforming labelled ASTs for program slicing

L-ASTs are the starting graphs of the EDG and they need to be properly transformed to be suitable for program slicing (Figure 5.10 shows the L-AST transformation of `y = 20` expression). The L-AST transformation basically consists in:

1. Add a new "*result*" node associated with each expression (i.e., each AST node labelled as *EXP*). It represents the value the expression is evaluated to during the execution.

2. All value edges are moved to the corresponding *result* nodes (e.g., the edge from `assign` to `y` now goes from the *result* of `assign` to the *result* of `y`). Furthermore, all outgoing control flow edges are moved to the associated *result* node (e.g., the edge from `assign` to `y` now goes from the *result* of `assign` to `y`) except for the *enter*/*exit* nodes, where the edges moved to the corresponding *result* node are the incoming control flow edges instead the outgoing ones. This way, the CFG has now two differentiated *enter* and *exit* nodes and control dependence algorithms can be correctly applied.

3. Add value and control flow edges from every expression to its corresponding *result* node. This value edge makes every *result* node dependent on the expression it represents. These three steps also prevent the loops that happen in other approaches [103, 107]. For instance, the loop between `assign` and `20` disappeared.

4. Remove all structural edges between expression nodes. This step removes syntactical intra-statement dependences that are not needed. For instance, in Figure 5.10, the structural edge between `assign` and `20` represents an unnecessary dependence because `20` is not influenced in any way by `assign`. Other approaches that keep the structural edges (e.g., [103, 107, 188]) employ them as a false dependence and this compromises the precision of their slices. In this particular example, they would produce the slice "`= 20`" when `20` is the slicing criterion.

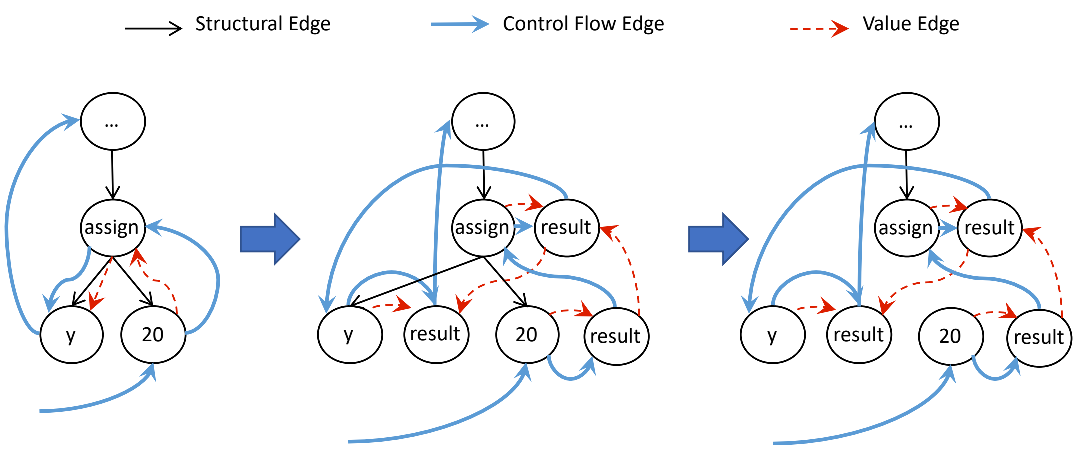


FIGURE 5.10: L-AST transformation for `y = 20`

This new *expression-result* structure is useful when traversing the graph because they divide node's responsibilities by separating each expression from its result. For instance, in the right graph of Figure 5.10 obtained from the transformation, selecting literal `20` does not include any other node in the slice because node `20` has no dependences in the graph. Note also that the original nodes are maintained in the transformation. This allows for selecting any expression without being interested in its value. For instance, selecting the `y` node as the slicing criterion does not include literal `20` in the slice. This is useful to reason about how the execution reached that point (control dependences needed to reach a particular point of the program). In contrast, if the value of `y` were of interest then its *result* node should be selected as the slicing criterion and, in such a case, literal `20` would be reached and included in the slice.

Algorithm 5.1 formalizes the transformation of the labeled AST. The auxiliary function $R(n)$ returns the result node of n. If n is not an expression node, then $R(n)$ returns n.

Algorithm 5.1 L-AST transformation

```
Input: A L-AST $G = (N, E)$ where $E = E_s \cup E_{cf} \cup E_v$.
Output: A transformed graph $G' = (N', E')$ where $E' = E'_s \cup E'_{cf} \cup E'_v$.
    ▷ Replace each expression node with a expression-result structure
 1: $N' \leftarrow N \cup \{R(n) \mid n \in N \wedge n \in \text{EXP}\}$
    ▷ Treat existing value edges $E_v$ and control flow edges $E_{cf}$
    ▷ $EE$ represents the set of enter/exit nodes of all procedures
 2: $E'_v \leftarrow \{(R(n), R(n')) \mid (n, n') \in E_v\}$
 3: $E'_{cf} \leftarrow \{(R(n), n') \mid (n, n') \in E_{cf} \;\wedge\; n \notin EE\}$
 4: $E'_{cf} \leftarrow E'_{cf} \cup \{(n, n') \mid (n, n') \in E_{cf} \;\wedge\; n \in EE\}$
 5: $E'_{cf} \leftarrow E'_{cf} \cup \{(R(n), R(n')) \mid (n, n') \in E'_{cf} \;\wedge\; n' \in EE\}$
    ▷ Add new value and control flow edges between expressions and results
 6: $E'_v \leftarrow E'_v \cup \{(n, R(n)) \mid n \in N \wedge n \in \text{EXP}\}$
 7: $E'_{cf} \leftarrow E'_{cf} \cup \{(n, R(n)) \mid n \in N \wedge n \in \text{EXP}\}$
    ▷ Remove structural edges in expressions
 8: $E'_s \leftarrow \{(n, n') \mid (n, n') \in E_s \wedge (n \notin \text{EXP} \vee n' \notin \text{EXP})\}$
```

Definition 5.5 (Transformed L-AST)**.** *A* transformed *L-AST (TL-AST) is the graph obtained as the output of Algorithm 5.1 with a L-AST as input.*

Theorem 5.1 (TL-AST completeness)**.** *Let G be a L-AST with a node $n \in G$. Let G' be its associated TL-AST (obtained by applying Algorithm 5.1 to G). If the slice s in G with respect to n is complete, then the slice s' in G' with respect to n is also complete.*

This theorem states that the TL-AST transformation does not produce any loss of *relevant* information, so that slices computed on the TL-AST are still complete. Note however that s and s' are not necessarily equal. This means that irrelevant information could be removed in s'. This will be stated later in a corollary of this theorem.

Recall that, in our context, a slice is complete if and only if it contains all the code that affects the values computed at the slicing criterion. Therefore, if we assume that value edges are correct (recall also that they are provided by the user), then the slice in G with respect to n is complete because the L-AST represents a single expression where all inter-dependences among nodes are given by structural (AST) edges (which we can assume correct because they are provided by the language) and value edges.

In order to prove Theorem 5.1, we need to consider the changes the TL-AST transformation introduces. These changes only affect expression nodes, so we build the proof over three independent lemmas according to the kind of edges implied in the transformation (namely, control flow edges, value edges, and structural edges). The rest of edges affecting the computation of the slice (control, flow, summary, etc.) are not affected by the transformation, so we do not need to consider them in the proof.

In the following lemmas and their corresponding proofs, we consider a L-AST as a graph $G = (N, E_{cf}, E_v, E_s)$, and its associated TL-AST as a graph $G' = (N', E'_{cf}, E'_v, E'_s)$. Therefore, $N \subseteq N'$. We also consider the set EXP of expressions in G, and function $R :: N \to N'$ where $R(n)$ returns the *result* node associated to node n (if the selected node is not an expression it returns the node itself).

The first lemma states that the TL-AST transformation does not affect the control flow. Roughly, the path of control flow edges followed in the L-AST is also followed in the TL-AST (only some *result* nodes are inserted into the path).

Lemma 5.1. *Control flow preservation. Given a L-AST, its control flow is preserved in its associated TL-AST: the traversal of the TL-AST nodes through control flow edges visits all nodes of the L-AST and in the same order as the traversal of the L-AST.*

Proof. First, every node in the L-AST is also present in the TL-AST ($N \subseteq N'$). Second, step (1) of Algorithm 5.1 includes a *result* node in N' for each expression node in N ($N' = N \cup \{R(n) \mid n \in N \land n \in \text{EXP}\}$). Moreover, in Algorithm 5.1, due to steps (3) and (7), we have that $\{\forall n, n' \in N, n \in EXP, n \notin EE \mid (n, n') \in E_{cf} \to (n, R(n)) \in E'_{cf} \land (R(n), n') \in E'_{cf}\}$, where EE is the set of *enter/exit* nodes of all procedures. Therefore, all the L-AST expression nodes connected to another node by a control flow edge are now transitively connected to the same node by means of their associated *result* node (except for *enter* nodes, which remain connected in the same way). No more connections are added in E'_{cf}, so all nodes in the L-AST are traversed in the TL-AST in the same order when we only follow control flow edges. □

The second lemma states that if a node $n' \in N$ is reached from node n via value edges in G, then, node $R(n')$ is also reached from node $R(n)$ via value edges in G'.

Lemma 5.2. *Value dependence preservation.* $\forall n, n' \in N \;.\; (n, n') \in E^*_v \to (R(n), R(n')) \in E'^*_v$, *where* E^*_v *(respectively* E'^*_v*) represents the transitive closure of* E_v *(respectively* E'_v*).*

Proof. First, there is a *result* node in N' for each expression node in N ($N' = N \cup \{R(n) \mid n \in N \land n \in \text{EXP}\}$). This is ensured by step (1) of Algorithm 5.1. Second, in Algorithm 5.1, due to step (2), we have that $\{\forall m, m' \in N, m, m' \in EXP \mid (m, m') \in E_v \to (R(m), R(m')) \in E'_v\}$. Hence, the nodes directly connected in L-AST by value edges are also has they *result* nodes directly connected in the associated TL-AST and the lemma trivially holds. □

The third lemma shows that structural edges between two expression nodes are redundant in the slicing phase due to value edges. Thus, removing them does not affect the completeness of the slices computed.

Lemma 5.3. *Unnecessary structural dependence. Let $G = (N, E_{cf}, E_v, E_s)$ be a L-AST with a node $n \in G$. Let $G' = (N, E_{cf}, E_v, E'_s)$ be a modified L-AST where $E'_s = \{(n, n') \mid (n, n') \in E_s \wedge (n \notin EXP \vee n' \notin EXP)\}$. If the slice s in G with respect to $n \in N$ is complete, then the slice s' in G' with respect to n is also complete.*

Proof. First, given a node n inside an expression e, the slice with respect to the slicing criterion n must contain all the nodes in e that may affect the value computed at n. Therefore, according to Definition 5.2, s and s' must contain all nodes that are value dependent on n.

We prove this lemma by induction on the size of e.

(Base case) First, in the case that e only contains one node, then the lemma holds trivially because there are no structural edges removed, and so $G = G'$. In the case that e contains two nodes (n and n', with $(n, n') \in E_s$), we have two possibilities:

1. If the structural edge is not removed, then $G = G'$ and the lemma holds.

2. If the structural edge is removed, then two cases exist:

   (a) $(n, n') \in E_v$: In this case, either if the slicing criterion is n or n', s' is complete because the path defined by the removed structural edge still exists thanks to the value edge.

   (b) $(n, n') \notin E_v$: In this case, when n' is the slicing criterion, s' is still complete even if n is not included in the slice (so, the structural edge can be removed) because the value of n is not required to compute the value of n' (otherwise, there would exist a value edge between them). Thus, s' is complete and the claim holds.

(Induction hypothesis) We assume as the induction that the lemma holds for an expression formed from i nodes.

(Inductive case) We prove that the lemma holds for any expression with $i+1$ nodes. Because the L-AST is a tree if we only consider structural edges, then, the new node m must be only connected to one node in e with a new structural edge.

Now, we have the following cases:

1. If the new structural edge is not removed, then $G = G'$ and by the induction hypothesis the lemma holds.

2. If the structural edge is removed, then let $m' \in N$ be the AST ancestor of m (i.e., $(m', m) \in E_s$). We only have to prove that m' is included in s' when it affects the value of m. Because value edges (Definition 5.3) are

the transitive reduction of value dependence (Definition 5.2), if m is value dependent on m' then $\exists(m', m) \in E_v^*$, where E_v^* represents the transitive closure of E_v. Therefore, s' is complete and the lemma holds.

□

Now we can prove the main theorem.

Proof. (Theorem 5.1). First of all, in order to preserve the execution paths necessary to compute later dependences, it is mandatory to fulfil that $\{\forall(n, n') \in E_{cf} \rightarrow (n, n') \in E'^*_{cf}\}$, where E'^*_{cf} represents the transitive closure of E'_{cf}. This is ensured by Lemma 5.1, which allows Algorithm 5.2 to compute the same dependences for both G and G'. Second, after the modifications to value dependence edges performed in Algorithm 5.1, the value dependences defined between the nodes in G must be preserved in the nodes of G'. The introduction of the *result* nodes in step (1), which stand for the value of their associated expressions, delegate this dependence to *result* nodes ($R(n)$) that are always value dependent of their associated expression node $\{\forall n \in N, n \in EXP \mid (n, R(n)) \in E'_v\}$. Hence, G' must fulfil that $\{\forall n, n' \in N \,.\, (n', n) \in E_v^* \rightarrow (n', R(n)) \in E'^*_v\}$, where E'^*_v represents the transitive closure of E'_v. This property is granted by Lemma 5.2. Finally, the removal of structural edges between expression nodes in G performed in Algorithm 5.1 must not affect the completeness of the slice s' computed over G'. Lemma 5.3 proves that the removal of those edges over G does not affect the completeness of the slice s with respect to any slicing criterion n. Since structural edges have no effect on any other edge modifications performed by Algorithm 5.1, their removal can be performed as a pre-processing of G or, as in Algorithm 5.1, at the end of the TL-AST transformation process. Either way this transformation does not affect the completeness of s'. Thus, all the steps of Algorithm 5.1 have been proven to preserve the dependences between the nodes of G (Lemma 5.1 and Lemma 5.2) and the completeness of the slice s' computed over G' (Lemma 5.3). □

Additionally, from the property proved at Lemma 5.3, we can derive the following corollary.

Corollary 5.1. *Precision improvement for the TL-AST. Let s be a slice computed on a L-AST $G = (N, E)$ with the slicing criterion $n \in N$, and let s' be the slice computed on the associated TL-AST G' with the same slicing criterion n. $s' \subseteq s$.*

Proof. (Sketch). If the TL-AST transformation does not remove structural edges from G, then, according to Lemma 5.2 and Lemma 5.3, $s = s'$. If the TL-AST transformation does remove structural edges from G, then the removed edges may produce s' to be smaller than s, in the case that the nodes connected with a path of structural edges (including the removed ones) in the L-AST is not connected also with a path of value edges (see the proof of Lemma 5.3). An example where $s' \subset s$ can be seen in Figure 5.10 (left and right) if we consider 20 as the slicing criterion. □

Equipping the graph with dependence edges

The TL-AST obtained by applying Algorithm 5.1 is ready to be equipped with the usual edges of program slicing, i.e., control, flow, call, input, output, and summary edges. In this section, we first explain how to equip the TL-AST with control and flow edges and then the resulting graph is adapted to interprocedural slicing.

The original control dependence definition (Definition 2.7) is still valid in the EDG because the properties in Definition 5.4 make the control flow edges in the TL-AST provide the same information as the one in the CFG. The flow dependence definition (Definition 2.8) considers *Definition-Use* dependences, and it is also valid in the EDG. Unfortunately, none of them account for variables that are declared and later defined. For instance, in the two first lines in the code of Problem 5.2 there does not exist a dependence between `(1,x)` and `(2,x)` captured by the SDG. Neither data nor control dependence exist because `(1,x)` is not a definition, but a declaration.[1] Therefore we provide a definition that considers *Declaration-Definition* dependences in typed languages.

Definition 5.6 (Declaration Dependence). *A node n_2 is* declaration dependent *on node n_1 if n_1 declares a variable x, n_2 defines x, and there exists a control flow path from n_1 to n_2.*

Algorithm 5.2 formally defines how the TL-AST is equipped with control, flow, and declaration edges.

Algorithm 5.2 Equipping TL-ASTs with control, flow, and declaration edges

```
Input: A TL-AST $G = (N, E)$ where $E = E_s \cup E_{cf} \cup E_v$.
Output: The graph $G$ becomes an intraprocedural EDG.
1: $E_c \leftarrow$ compute control dependence normally (Definition 2.7).
   ▷ FD stands for all pairs of nodes that are flow or declaration dependent. DEF and USE
   stand for all nodes that are definitions or uses, respectively
2: $E_d \leftarrow \{(n, n') \mid (n, n') \in \text{FD} \land n' \in \text{DEF}\}$
3: $E_f \leftarrow \{(R(n), R(n')) \mid (n, n') \in \text{FD} \land n' \in \text{USE}\}$
   ▷ Control flow edges are no longer needed
4: $E \leftarrow E_s \cup E_v \cup E_c \cup E_d \cup E_f$
```

In line 1, control edges are computed as in the SDG. Lines 2 and 3 compute declaration edges (E_d) and flow edges (E_f). The declaration edges generated for E_d leave from n (but not from $R(n)$), which ensures that the initialisation of a variable is avoided when only its declaration is needed.

Example 5.3 (Control and declaration edges). *An example of control and declaration edges generated by Algorithm 5.2 is shown in Figure 5.11.*

In the figure, there are six control edges. The declaration edge at the bottom is generated from the E_d set in Algorithm 5.2. Because this edge leaves from

[1]The SDG needs a post-process to identify (and include) the declarations of all variables included in the slice (`(1,x)` would be included in that post-process). This post-process solves the problem of making the slice well-formed (and executable). However, the fact of not explicitly representing declarations in the SDG prevents SDG-based analyses to handle this information at slicing time, for instance to properly handle polymorphism.

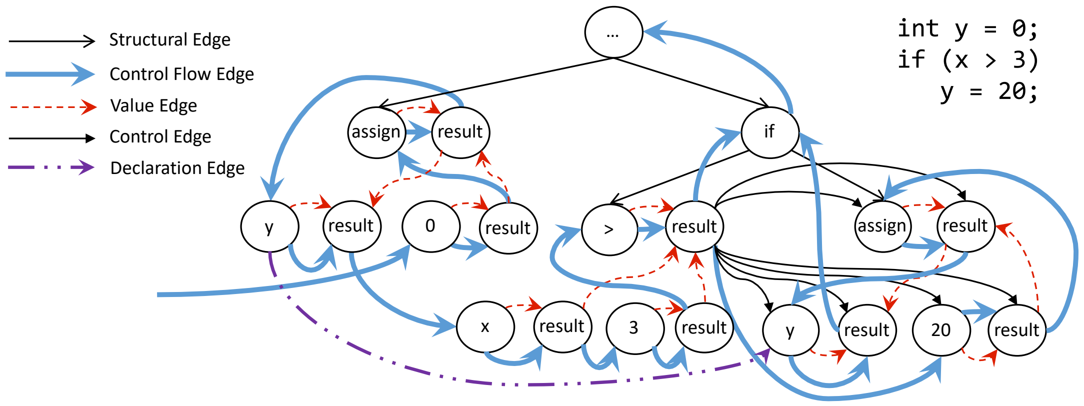


FIGURE 5.11: Control and declaration edges generated by Algorithm 5.2

the y node instead of from its result *node, when the definition of y on the right is included in the slice, its declaration on the left is also included but its initialisation ignored.*

The intraprocedural EDG obtained by applying Algorithm 5.2 now can be augmented to treat interprocedural programs. Traditionally (see Section 2.1), this is accomplished by (i) creating fictitious assignments for argument and parameter nodes and then (ii) adding call, input, output and summary edges. In our approach, (i) we naturally (by construction) represent arguments and parameters in separate nodes without creating fictitious assignments, so the call graph is enough to match the arguments and the parameters considering their positions in each call/definition, and (ii) all edges are calculated in the same way as in the SDG approximation. The main difference between the interprocedural edges construction methods is that we connect *result* nodes instead of variable nodes. Therefore, input (respectively output) edges connect the *result* of actual-in (respectively formal-out) nodes with the *result* of formal-in (respectively actual-out) nodes. To handle values returned by functions, the functions are considered expressions (with their corresponding *result* node) and an output edge connects this *result* node with the *result* node of the corresponding calls. Finally, summary edges are added to the graph, connecting the *result* node of the necessary parameters to the *result* node of the call, and slices are obtained in exactly the same way as in the SDG approximation.

There is an interesting aspect in the TL-AST of Figure 5.11 related to types that is worth mentioning. Note that in this specific TL-AST there does not exist a node that explicitly represents the type `int`. This is not a problem, it is just a possible AST representation. In fact, because the EDG is multi-paradigm, it is able to represent typed and untyped languages. This is one of the advantages of labelling the AST with extra information, including what nodes are expressions, declarations, definitions, or uses. In this particular case, the left `y` node has been labelled as a declaration with its type, and the right `y` node has been labelled as a definition; and this is why there is a declaration edge connecting them.

An EDG for the interprocedural code in Figure 5.12 is shown in Figure 5.13. In this EDG, actual-out and formal-out nodes have been added for clarification purposes. However, as it occurs in the SDG, they can be omitted if the underlying variable cannot be modified inside the procedure.

```
int x = 0, y = 5;
x = 10;                                  boolean foo(int a, int b) {
if (foo(x, y)) {                             return a == 10 | b == 15;
    x = y = 20;                          }
}
```

FIGURE 5.12: An interprocedural version of the code in Figure 5.2

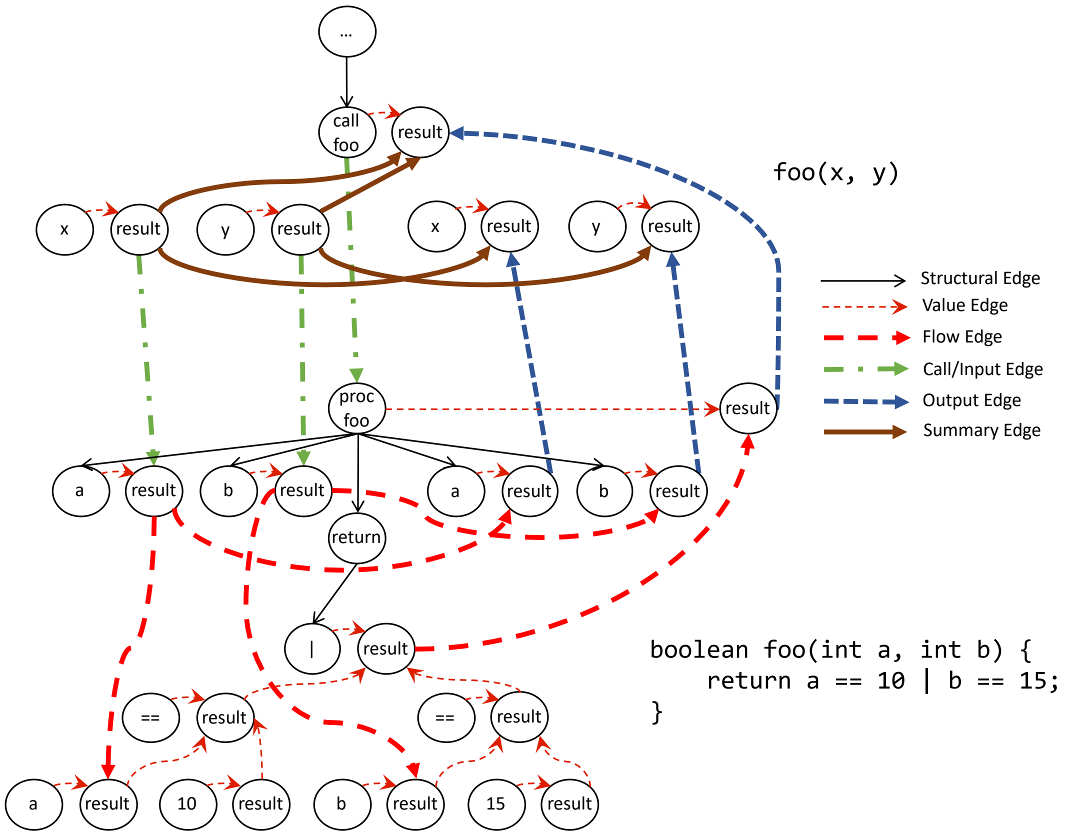


FIGURE 5.13: An EDG for the code in Figure 5.12

Now that we have presented the SDG and the EDG, we can state an interesting result: the EDG is a generalisation of the SDG.

Theorem 5.2 (SDG generalisation)**.** *Given a SDG, there always exist an equivalent EDG with the same nodes and edges.*

Proof. We prove this theorem by showing that an SDG is an instance of an EDG where all nodes are statements (no expression nodes exist). Let P be a program, CFG_P be the control flow graph of P, and (N, E) be the SDG of P. First, we can always construct an AST of P, AST_P, where the AST nodes (N') are only the statements in P. Therefore, $N = N'$.

Second, the $L\text{-}AST_P$ is the AST labelled with control flow edges and value edges. Because $N = N'$, no SDG node has been broken down, and thus no value edge exist because value dependence only appears when breaking down an SDG node (value edges represent intra-statement dependences).

Third, because $N = N'$ and $L\text{-}AST_P$ is the AST labelled with control flow edges, then for each procedure p of P, CFG_p is equal to $L\text{-}AST_p$ if we just ignore the structural edges. Moreover, if we consider all AST nodes as statements ($EXP(N') = \emptyset$), then the transformation to $TL\text{-}AST_P$ would not contain value edges either because only expressions contain value edges according to Algorithm 5.1, and no node would be decomposed. This is true because only expression nodes are exploded and equipped with additional *result* nodes. Therefore, $L\text{-}AST_P = TL\text{-}AST_P$ if we ignore the declaration edges. Finally, to build the EDG from $TL\text{-}AST_P$ we would need to add:

Control dependence edges: According to Algorithm 5.2, they are the same in the SDG and the EDG because $N = N'$ and $CFG_P = TL\text{-}AST_P$.

Flow edges: According to Algorithm 5.2, they are equivalent because $N = N'$ and, thus, $DEF(N) = DEF(N')$ and $USE(N) = USE(N')$.

Call / input / output edges: They are equivalent because $N = N'$ and they use the same algorithm to be constructed.

Summary edges: They are equivalent because they use the same algorithm to be constructed based on the previous three sets of edges, which are equivalent in both graphs. Therefore, the same paths would be summarised.

Because $N = N'$ and all edges are equivalent, then $SDG_P = EDG_P$.

□

5.3 Slicing the EDG

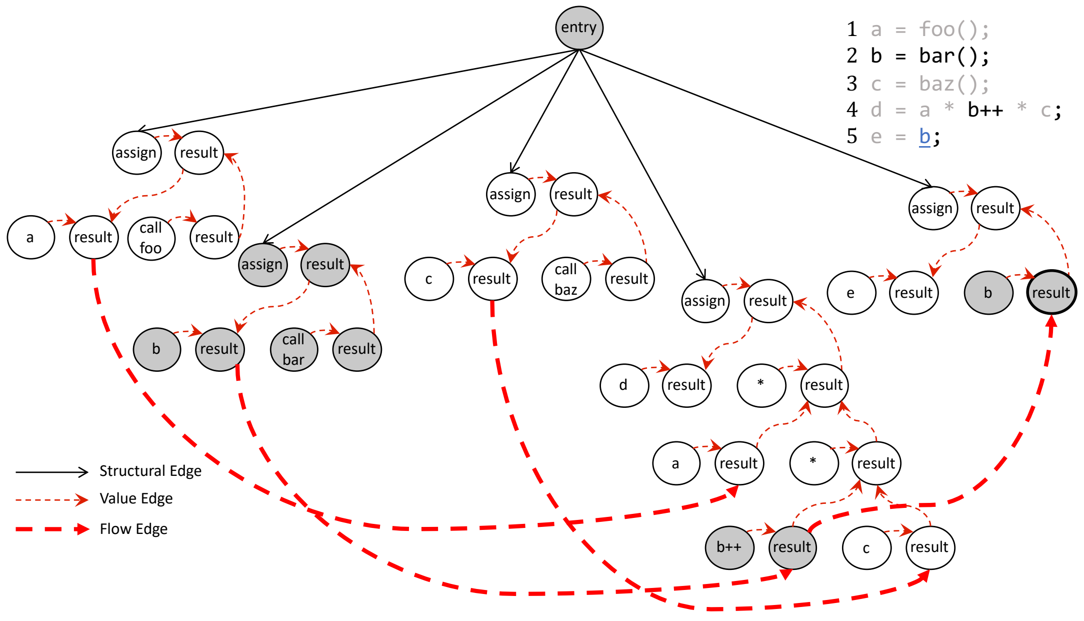
The algorithm that slices the EDG is exactly the same as the algorithm that slices the SDG. The classical Weisser's algorithm [204], the interprocedural algorithm by Horwitz et al. [85], and later improvements such as [16, 169] are all of them directly applicable to the EDG. This is the beauty of the EDG: it follows (and extends) the same principles of the SDG: a backward traversal of the EDG from the slicing criterion produces a complete well-formed slice. Moreover, all additional slicing restrictions used to improve the precision of slices for particular program structures (like the one introduced for the slice of unconditional jumps [109] or exception handling [62]) can be directly applied to the EDG in the same way they are to the SDG. Other techniques, nevertheless, need to define ad-hoc algorithms for their extensions. For instance, the fine-grained graph representation defined by Krinke and Snelting [103, 107] introduced some traversal restrictions in the slicing algorithm to avoid including in the slice some irrelevant subexpressions (e.g., some operands in binary operations —see the related work section—).

5.4 Solving SDG limitations

This section shows how the five previously described problems are solved by the EDG without making any kind of special treatment to any language structure. In all the figures, the slicing criterion is represented with a bold-border node, and all the nodes in the slice are represented in grey.

Extraction of embedded definitions

Figure 5.14 shows the EDG of the scenario shown in Problem 5.1. In the EDG, the assignment in line 4 is represented with 14 different nodes, where only the 2 nodes representing the definition of variable `b` in `b++` are included in the slice, ignoring the rest of the assignment nodes.



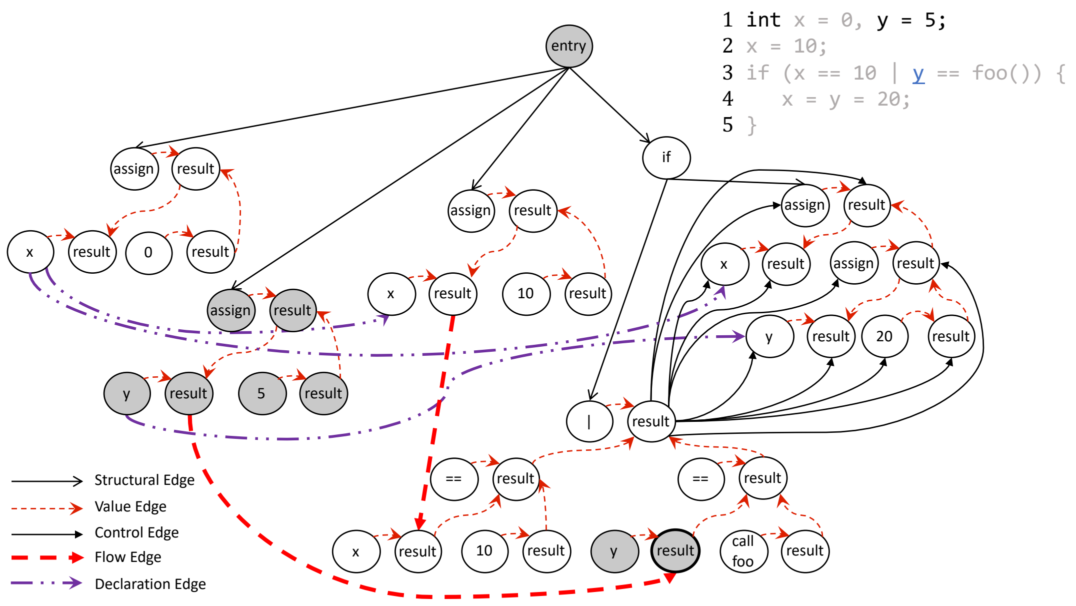
```
1 a = foo();
2 b = bar();
3 c = baz();
4 d = a * b++ * c;
5 e = b;
```

FIGURE 5.14: An EDG for the code in Problem 5.1 and slice w.r.t. $\langle 5, b\rangle$.

The way value dependences are defined inside the assignment of line 4 allows the graph to include inner subexpressions (the `b++` subexpression) preventing to include the outer expression (the whole assignment). Additionally, flow dependences include in the slice the previous definition of `b`, necessary to compute the new one. This fact leads to the expected slice (Figure 5.1b), shown with grey nodes in Figure 5.14.

Monolithic code

Figure 5.15 shows the EDG for the code in Figure 5.2 that solves the problem presented in Problem 5.2. In the EDG, because structural edges are removed inside expressions, the `if` node and the other elements of the `if` predicate are correctly ignored. Moreover, the declaration and initialisation of variable `y` in line 1 are included thanks to the flow edge.



```
1 int x = 0, y = 5;
2 x = 10;
3 if (x == 10 | y == foo()) {
4     x = y = 20;
5 }
```

FIGURE 5.15: An EDG for the code in Problem 5.2 and slice w.r.t. $\langle 3, y \rangle$

try-catch structures

Figure 5.16 contains the EDG of the code given in Problem 5.3. This EDG shows how a `try-catch` statement is represented by our graph. The key of our model is the representation of the assignment in line 4. The right-hand side of this assignment is the potential source of error that controls the whole `catch` block. The structure that represents this statement makes use of variable `b` in line 6 exclusively dependent on the definition of `b` in line 2, excluding the definition of `b` in line 4 from the slice. Finally, the declaration edge from the definition of `b` in line 2 to the declaration of `b` includes also line 1 in the slice.

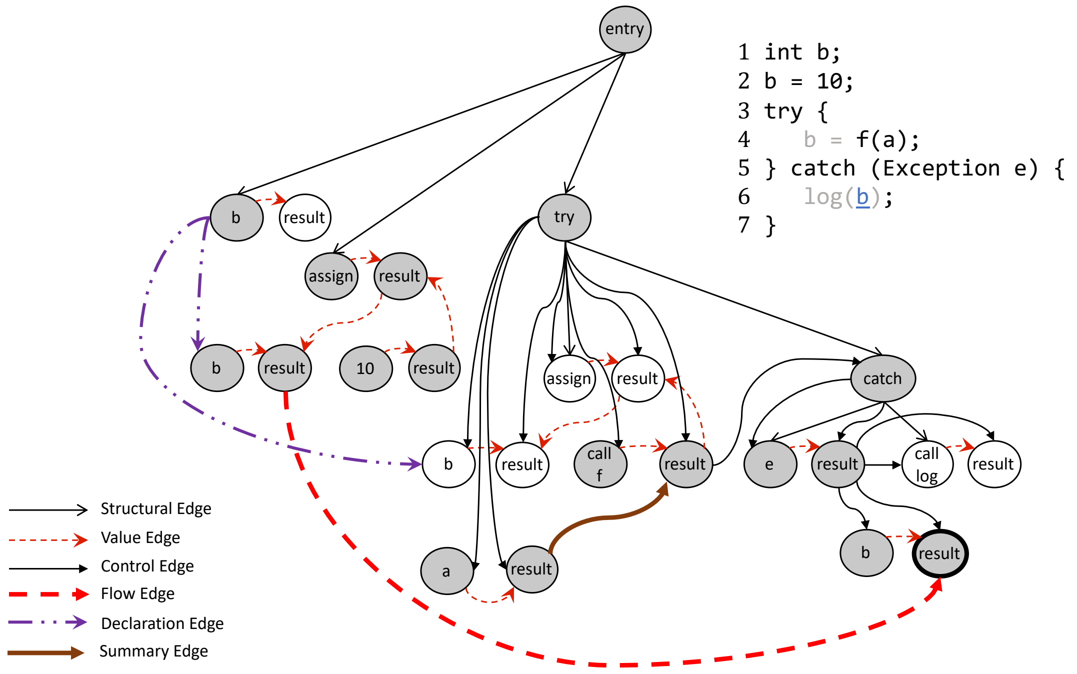
for loops

The EDG also gives a new representation for `for` loops. Figure 5.17 shows this new representation. In the EDG, the loop node is structurally connected to each one of its components, and there are control dependences between the condition and the body and update blocks. Additionally, each block is divided into multiple statements which, in turn, are composed of a set of nodes.

In this case, the value of `x` selected as the slicing criterion, only depends on the `x` definitions in the initialisation and update blocks. Since the update block is represented by different nodes, it is trivial to exclude the `it` variable and all its dependences from the slice.

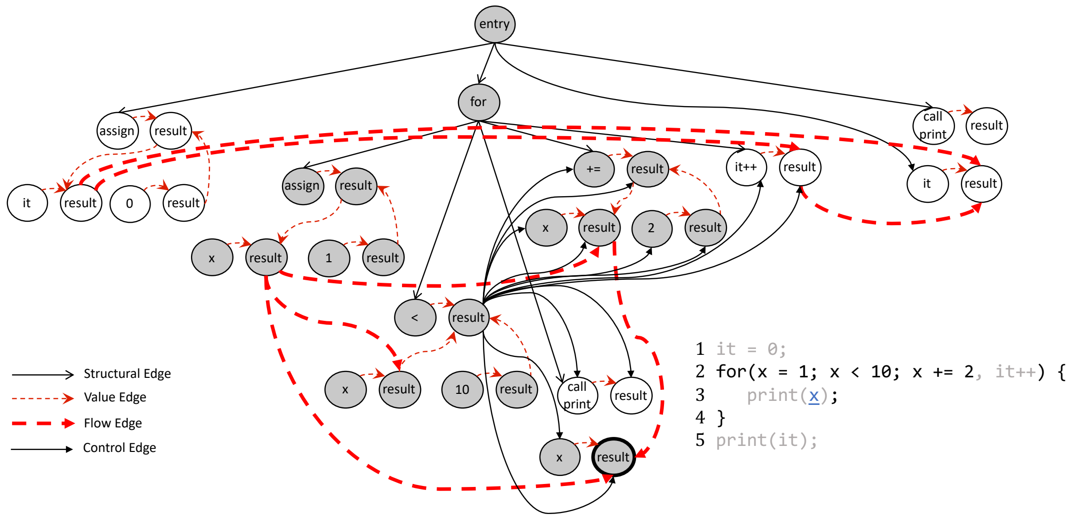
List comprehensions

This EDG in Figure 5.18 shows the representation of a common structure in functional programming, the list comprehension. In a list comprehension, each component is control dependent on the previous one because even the pattern



```
1 int b;
2 b = 10;
3 try {
4     b = f(a);
5 } catch (Exception e) {
6     log(b);
7 }
```

FIGURE 5.16: An EDG for the code in Problem 5.3 and slice w.r.t. ⟨6, b⟩



```
1 it = 0;
2 for(x = 1; x < 10; x += 2, it++) {
3     print(x);
4 }
5 print(it);
```

FIGURE 5.17: An EDG for the code in Problem 5.4 and slice w.r.t. ⟨3, x⟩

matching executed as the first element may prevent the evaluation of later executed components (other generators, filters, and the expression). In this case, the slicing criterion is located in the generator, so there is no need to include in the slice components that are executed after it, like filters or the expression. Additionally, since the slicing criterion is a single element of a tuple expression, the granularity of the EDG allows us to ignore the rest of elements, including in the slice only the container expression (the tuple signature) and the variable that provides the values at the generator (`L1`).

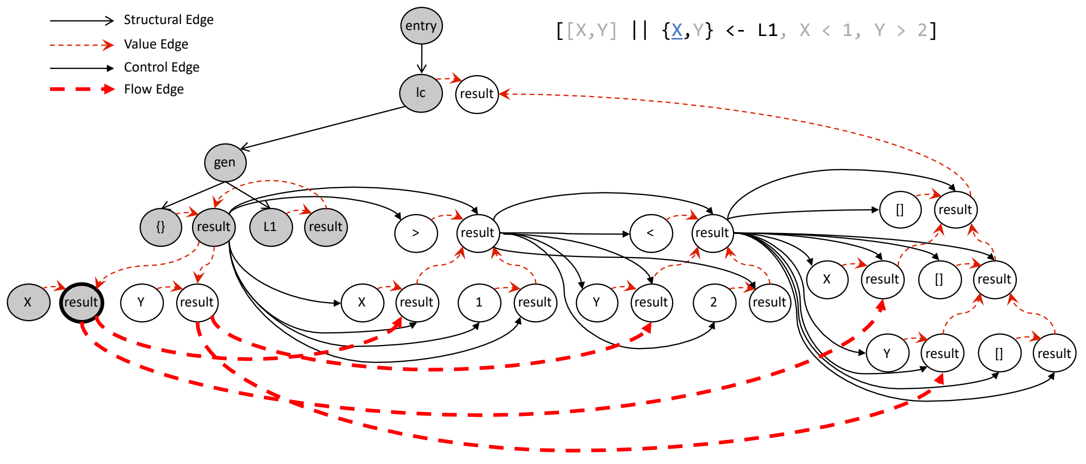

```
[[X,Y] || {X,Y} <- L1, X < 1, Y > 2]
```

FIGURE 5.18: An EDG for the code in Problem 5.5 and slice w.r.t. $\langle 1, X\rangle$

5.5 Implementation

All the algorithms described in this paper have been implemented in a program slicer called e-Knife. e-Knife is a program slicer that operates over the sequential part of the Erlang language. e-Knife is implemented in Java and operates with the EDG as its base representation graph. Given an Erlang program and a slicing criterion, e-Knife generates the corresponding EDG and it slices it with a high level of precision. The e-Knife project is composed of two different modules: *e-Knife* and *EDG*. While the *e-Knife* module represents the language-dependent part of the EDG model including parsing, unparsing, and CFG building, among other processes; the *EDG* module represents the language-independent part, including dependences computation and slicing algorithms. Counting both modules, the implementation is composed of more than 9600 lines of code, divided into 51 Java files classified in 13 different packages. All the source code is publicly available at `https://mist.dsic.upv.es/git/program-slicing/e-knife-erlang` and a limited demo to try the tool can be found at `https://mist.dsic.upv.es/e-knife-constrained/`.

Figure 5.19 shows the most relevant modules and classes of e-Knife, and how they are connected in order to compute a slice from a given Erlang program. In the figure, we differentiate two main modules in the implementation (represented with rounded squares): the module that specifies the language-dependent part of the slicer (eKnife) and the module that specifies the language-independent part of the slicer (EDG). Additionally, in Figure 5.19 squares represent Java classes inside these modules, solid arrows represent calls between classes (sometimes between different modules), and dashed arrows represent the input/output resources of the slicer.

Although the implementation contains 51 classes, Figure 5.19 only shows the communication between the most relevant 13 to give a general idea of the whole process. Each one of these classes and the functionality they provide is the following:

- `EKnife`. This class manages the whole slicing process. It is the class that

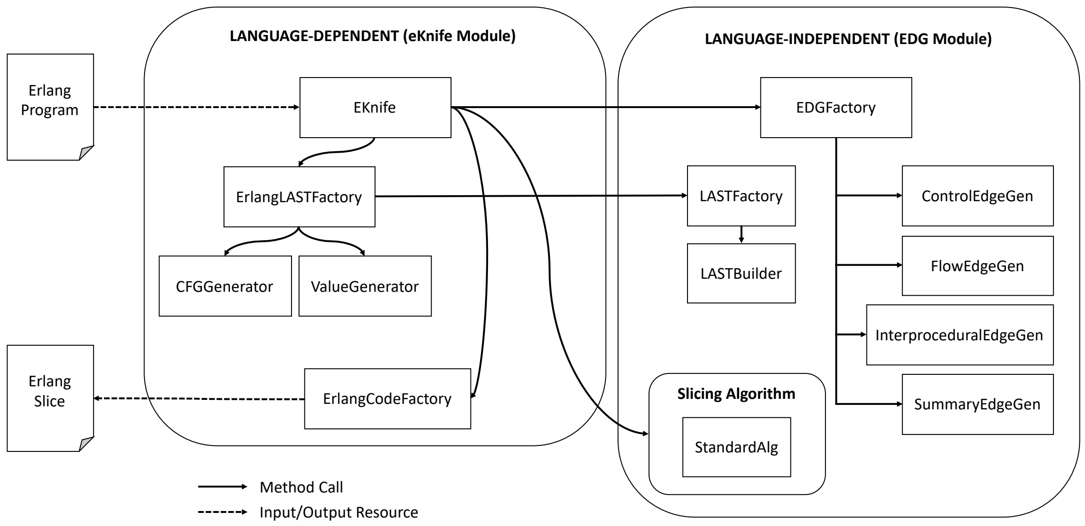


FIGURE 5.19: `e-Knife` architecture and communication between modules

calls to all the other classes passing the results they provide to the next step of the process (i.e., after creating the LAST, this class passes it to the `EDGFactory` class to create the TLAST and the final EDG).

- `ErlangLASTFactory`. This class is the one that manages the process to build the LAST. First, the class extracts the AST of the given program and transforms this AST (usually contained in a recursive sequence of elements) to an internal graph-like representation by successively calling methods defined at the `LASTFactory` class. This class additionally classifies each AST element it processes as declaration, definition, use, and/or expression to form the so-called *DEC*,*DEF*,*USE*, and *EXP* sets. Finally, control flow and value dependences are given to the created AST by traversing the AST with a call to the `CFGGenerator` and `ValueGenerator` classes respectively, obtaining the LAST.

- `LASTFactory`. The `LASTFactory` class is in the language-independent module that works with generic elements. The idea is for the language-dependent class (`ErlangLastFactory` in this case) to extend this Java class. This class gives a generic treatment to the AST provided by a specific language. It contains methods that classify an AST element, creates its associated new nodes in the internal graph representation (through calls to `LASTBuilder`), joins these nodes with structural dependences, and recursively processes the next AST subelement.

- `LASTBuilder`. This class creates the nodes in the internal graph and correctly connects them to their corresponding AST parent. Its methods receive a type of node and any extra information important to store inside the node for dependence computation, e.g., whether it is a definition or a use, or whether it is an expression or not.

- `CFGGenerator`. This generator class is responsible for generating a CFG over the AST previously computed in `ErlangLASTFactory`. The CFG is dependent on the language being treated, because the flow of the execution may change according to certain features of the language, e.g., eager or lazy evaluation.

- `ValueGenerator`. This class traverses the AST generated in `ErlangLASTFactory` and includes in the graph the value dependences, which are computed according to the execution and semantics of the programming language being treated.

- `EDGFactory`. The `EDGFactory` class implements Algorithms 5.1 and 5.2. To implement Algorithm 5.2 it makes use of four different Java classes: `ControlEdgeGen`, `FlowEdgeGen`, `InterproceduralEdgeGen`, and `SummaryEdgeGen`.

- `ControlEdgeGen`. This class receives a LAST and it uses its CFG to compute control edges using the post-dominator tree algorithm [154].

- `FlowEdgeGen`. This class receives a LAST and uses the CFG and the *DEC*, *DEF*, and *USE* sets (information contained inside each LAST node) to traverse the graph and compute declaration and flow edges.

- `InterproceduralEdgeGen`. This class receives a LAST and generates the input and output edges by computing a call graph and linking each method call to the possible called method definition.

- `SummaryEdgeGen`. This class computes the summary edges by applying the intraprocedural slicing algorithm for each method definition that can be reached in the call graph.

- `StandardAlg`. This class implements the interprocedural slicing algorithm proposed by Horwitz (it is shown in Section 2.2 of Chapter 2).

- `ErlangCodeFactory`. This class is language-dependent and it is responsible for transforming the resulting sliced EDG back to Erlang code, traversing the EDG and excluding from the code transformation those nodes that are not part of the slice.

Despite explaining only the main 13 classes, the rest of Java classes are part of the EDG's infrastructure and support each process performed by these 13 classes. Some of them represent other implementation requirements like the inner structure of the graph (`Node`, `Edge`, `GraphWithRoot`, `LAST`, `EDG`...), the representation of key elements like the slicing criterion (`SlicingCriterion`), or the visual representation of the graph (`DotFactory` and `PdfFactory`), among other features. The whole project with all the source code implemented can be found in the Github repository:

`https://mist.dsic.upv.es/git/program-slicing/e-knife-erlang`

5.6 Empirical evaluation

We conducted experiments to empirically evaluate the precision and performance of the new slicing library and program slicer. These experiments provide a precise and quantitative idea of the performance of the EDG compared to the SDG.

To establish a goal for the empirical evaluation, we ask two research questions, comparing the EDG to the SDG:

RQ1. *What are the resource requirements required to create a slice with the EDG compared with the SDG?* This includes both generation and slicing time requirements. To compare this values across larger and smaller benchmarks, we compute the ratio between both graphs (t_{edg}/t_{sdg}) in both parts of the process.

RQ2. *What is the improvement in precision introduced by the EDG?* Given that we are comparing techniques over different graph representations, we consider that the fairest comparison would be to compare the difference between the sizes of the computed slices' ASTs. For this reason use the formula $(A-B)/A$ where A is the size (number of AST nodes) of the slice computed over the SDG and B is the size (number of AST nodes) of the slice computed over the EDG.

Benchmarks and experiment design

In order to compare both implementations we have chosen Bencher [152], a program slicing benchmark suite for Erlang. It contains 18 Erlang programs, in which a variety of structures and program constructs can be found.

For each benchmark we run and measured the following steps: (1) generate the EDG and the SDG, and (2) for every slicing criterion, slice the graph. To create a fair selection of slicing criteria to adequately compare the EDG and the SDG, we use as slicing criteria each variable in each line of the program. In the EDG, this means selecting the *result* node of that variable; and in the SDG, this is equivalent to a classic slicing criterion $\langle l, v\rangle$, which selects the whole statement as criterion. This procedure is more fair and representative than selecting an arbitrary slicing criterion because it allows us to answer both research questions w.r.t. any situation, and thus obtain valid average metrics for each benchmark. Moreover, this methodology avoids that the arbitrary selection of one slicing criterion benefits one of the program representations (EDG/SDG).

RQ2 only requires each combination of benchmark and slicing criterion to be run once, as it compares the resulting slices against the original program and the slicing process is deterministic. On the other hand, RQ1 requires running each combination repeatedly, in order to obtain the average consumption of time and memory.

All experiments were run on a 1.4 GHz Quad-Core Intel Core i5 processor with 8 GB RAM 2133 MHz LPDDR3 under macOs Version 11.5, while all other non-critical processes were stopped. We strictly followed the methodology proposed in [67]. Each operation (building or slicing a graph) was performed

File	**Graph Generation**		**Slicing**			
	t_{EDG}	t_{SDG}	# SCs	Prec. Inc. (%)	t_{EDG}	t_{SDG}
bench1.erl	3310.33ms	1823.74ms	284	$1.90 \pm 0.95\%$	2038.09μs	249.21μs
bench2.erl	82.94ms	47.30ms	66	$23.47 \pm 6.13\%$	93.96μs	37.48μs
bench3.erl	34.79ms	18.85ms	25	$19.88 \pm 9.32\%$	40.08μs	16.65μs
bench4.erl	30.72ms	20.19ms	39	$36.33 \pm 6.77\%$	81.53μs	24.18μs
bench5.erl	17.52ms	10.46ms	17	$14.71 \pm 8.00\%$	46.02μs	20.14μs
bench6.erl	60.29ms	33.58ms	54	$28.00 \pm 8.00\%$	40.09μs	17.73μs
bench7.erl	7.03ms	4.12ms	12	$11.41 \pm 9.00\%$	30.54μs	14.61μs
bench8.erl	100.29ms	55.16ms	46	$7.92 \pm 1.73\%$	207.04μs	61.78μs
bench9.erl	37.92ms	22.33ms	26	$4.74 \pm 2.94\%$	132.46μs	33.36μs
bench10.erl	133.62ms	82.04ms	55	$5.83 \pm 3.25\%$	257.03μs	75.40μs
bench11.erl	10.05ms	6.01ms	18	$15.48 \pm 6.54\%$	36.37μs	15.79μs
bench12.erl	515.01ms	365.00ms	142	$8.12 \pm 3.30\%$	700.97μs	228.78μs
bench13.erl	28.55ms	17.08ms	28	$3.62 \pm 2.85\%$	129.21μs	47.47μs
bench14.erl	175.78ms	98.50ms	61	$35.83 \pm 7.96\%$	98.21μs	34.92μs
bench15.erl	276.32ms	158.93ms	77	$46.36 \pm 7.29\%$	216.28μs	71.57μs
bench16.erl	119.00ms	65.18ms	42	$19.99 \pm 9.27\%$	122.55μs	45.19μs
bench17.erl	65.83ms	36.73ms	25	$0.26 \pm 0.50\%$	167.44μs	65.54μs
bench18.erl	74.83ms	43.36ms	29	$0.17 \pm 0.34\%$	271.28μs	103.11μs
Average	**282.27ms**	**161.59ms**	**1046**	**14.20±4.12**	**729.71μs**	**126.39μs**

TABLE 5.1: Results of the experimental evaluation comparing the SDG and the EDG

1001 consecutive times (over one million slices in total). To ensure real independence, the first iteration was always discarded (to avoid influence of data and code caches). From the 1000 remaining iterations we retained all windows of 10 measurements where steady-state performance was reached, i.e., once the coefficient of variation (CoV, the standard deviation divided by the mean) of the 10 iterations falls below a preset threshold of 0.01. In all cases we selected the steady-state performance window whose average was closer to the mean of the complete sample. In some benchmarks 0.01 was difficult to reach in 1000 iterations; so in those cases, we took the window of 10 iterations with lower CoV. Then, for each pair benchmark-criterion, we computed the mean of the 10 iterations under steady-state performance. We collected this mean for each criterion of a benchmark to produce the mean and standard deviation for said benchmark.

Results

This process was repeated for the 18 benchmarks, totalling 1046 different slicing criteria. The precision and time measurements collected are shown in Table 5.1.

Each benchmark is identified by columns `Benchmark` and `File`. The rest of the table is divided between the *generation* and *slicing* of each graph. In the generation, there are two columns showing the average time required to build each benchmark's EDG and SDG computed with the steady-state performance window described before. The *slicing* part of the table shows the number of slicing criteria (`#SCs`) selected for that benchmark; the average precision increase with 95% error margins[2] (`Prec. Inc.`), as described in RQ2; and the average time (also computed with the steady-state performance window) required to produce a slice in each graph.

To answer RQ2, we must look at the average precision improvement for the EDG, which is 14.20%. This means that we can expect EDG slices to be 14.20% smaller than their SDG counterparts. Regarding RQ1, the EDG requires 5.77 times more time to produce a slice; which is expected, considering that the EDG is bigger than the SDG. However, the extra time required for this improvement is very small considering that times are measured in microseconds: the average slicing time in the EDG (a linear cost operation) lasts, in average, 0.6 milliseconds more (up to 1.8 milliseconds in the worst case scenario) than doing so from the equivalent SDG. Constructing the EDG (a quadratic cost operation) lasts, in average, 121 milliseconds (up to 1500 milliseconds in the worst case scenario) more than constructing the SDG, but this cost is traditionally less important, because once a graph has been built, any number of slices can be extracted from it.

If we look again at the precision improvement, we can observe in the error high error margins (up to 9.32% in `bench3.erl`). This is due to the differences in the slicing criteria selected: some slicing criteria produce the same slice (thus, the improvement is zero), other slicing criteria, however, produce a big improvement (e.g., because just some subexpression is needed and the EDG can just pick it, without including the full statement). This confirms something expected: the different slicing criteria has a very big impact on the slices.

It is worth mentioning that, error margins of the time required to slice are big. This can be explained easily: some slicing criteria produced big slices with its corresponding longer traversal time, while other slicing criteria produced very small slices (e.g., the slice of the first variable in a function).

5.7 Related Work

The idea of a fine-grained SDG is not new. In fact, it is almost as old as the PDG itself, since it was already discussed by Ottenstein and Ottenstein in [147]. However the first fine-grained program slicing technique was proposed one decade later by Ernst [55]. Ernst proposal, however, was not based on the PDG/SDG, but on another program representation called the Value Dependence Graph (VDG) [203]. Soon, Kinloch and Munro [97] presented several ideas about how to do so in the PDG/SDG. They proposed to create extra graph vertices whenever a sub-expression contains side effects or control flow. Later on, Sloane and Holdsworth [182] proposed to use annotated ASTs to slice expressions. Their approach was pioneer and has the same spirit as our work. The same idea was explored by Steindl [188], who coined the term "*expression-oriented slicing*" (in contrast to "*statement-oriented slicing*"). However, the technique by Steindl was specifically focussed on object-oriented programs. He

[2]Recall that the EDG is always as least as precise as the SDG, thus even if the error margins suggest otherwise, the minimum increase is always zero.

described how to solve particular problems of that paradigm (polymorphism, encapsulation...) but a description of how to annotate the ASTs was missing.

Krinke and Snelting [103, 107] took a step forward in the same direction and they proposed a general extension of the AST with the introduction of new dependences: the *immediate (control) dependence* (in `x+y`, `x` and `y` are immediate dependent on `+`), a kind of intra-expression data dependence called *value dependence*, and the *reference dependence* to represent variable assignments. This work is a source of inspiration and is the basis of our work. In fact, our notion of value dependence comprises their value and reference dependence; and their immediate dependence is very similar to our structural dependence (both represent the AST structure). With respect to the slicing algorithm, due to the existence of dependence loops, they had to modify it: they do not traverse value dependence edges if the actual node has been reached by an immediate dependence edge and vice versa. Our work further generalizes this approach with various extensions, being the most important the introduction of the *result* node. Apart from the already explained advantages, result nodes together with the removal of the structural dependences (the AST structure) in expressions prevent the occurrence of loops, and thus, the necessity of changing the slicing phase.

One important difference with [107] and [103] is that we explicitly represent the value flow between expression components. Moreover, in contrast to them, where only variable nodes can be used as slicing criteria, in our work (and also previously in [188]), any AST node can be the slicing criterion. This increases the slicing capabilities and is particularly useful, e.g., for testing program slicers [152].

Another important advantage of our approach is that it can work for any AST (it is language-independent), as it has been shown with our EDG instantiations for Java and Erlang (see the appendices). The whole EDG structure depends on the control flow defined over the AST expressions, but not on the AST itself, which can have any tree structure. Therefore, if the control flow is provided all of our dependences can be constructed. We think that this is the natural extension of the SDG, which also defines its dependences over the statements control flow.

It is worth mentioning the work by Thomas Reps et al. [167, 169] to increase the precision of the SDG. Their work relies on a different extension of the SDG called the IFDS/IDE framework, and it was focussed on making program slicing context-sensitive [1, 76, 105, 106] and field-sensitive [72, 123]. Their extensions are based on the original SDG and roughly consist on the labelling of the edges with information about calling context and about fields in data structures. Therefore, this work also reduces the granularity of the slices in the case of composite data structures. Unfortunately, many expressions and program constructs cannot be handled with the propagation of fields. Their work is orthogonal to ours, and it could be applied to the EDG. It would be particularly interesting to apply the IFDS/IDE edges labelling to the EDG's intra-statement dependence edges. Both techniques are complementary and their combination could produce an important synergy.

We believe that the EDG can open a door to the application of slicing techniques in the declarative paradigm. Program slicing has been traditionally associated with the imperative paradigm. In fact, practically all slicing-based techniques have been defined in the context of imperative programs and very few works exist for declarative languages (notable exceptions are [57, 144, 168]). However, the SDG has been adapted to other paradigms such as the object-oriented paradigm [112, 118, 202] or the aspect-oriented paradigm [216].

The main problem when trying to adapt the SDG to a declarative language is that the SDG is based on statements, while declarative languages are often based on the composition of expressions. For instance, many Haskell programs are composed of a single statement formed by the composition of several expressions:

```
quicksort (x : xs) = quicksort [a | a <- xs, a <= x]
  ++ [x] ++ quicksort [a | a <- xs, a > x]
```

In this Haskell program, no slicing is possible with the PDG/SDG. In contrast, the EDG defines flow dependences among the subexpressions that allow us to extract slices without the need to transform the source code.

There have been previous attempts to define a PDG-like data structure for functional languages. The first attempt to adapt the PDG to the functional paradigm was [171] where they introduced the *functional dependence graph* (FDG). Unfortunately, the FDGs are useful at a high abstraction level (i.e., they raise the granularity level of nodes to functions and, therefore, they can only slice (complete) modules or functions), but they cannot slice expressions and thus they are insufficient for, e.g., Haskell or Erlang. Another related approaches are the *term dependence graphs* (TDG) [32], the approach in [25] (that uses the AST of Haskell), and the *behavior dependence graphs* (BDG) [199]. They are specific for term rewriting systems, Haskell, and Erlang, respectively. Unfortunatelly, they lack important features of the SDG as other approaches (e.g., [205, 206]). For instance, most of them lack of summary edges [85], which prevents them to be precise in the interprocedural case. The most advanced adaptation of the SDG to a functional language are the Erlang Dependence Graphs [180]. This is a notable approach that breaks down nodes to represent every single expression in the program (we were also inspired by this work and we achieve the same minimal granularity level). Unfortunately, these graphs are specific for Erlang.

The idea of applying slicing for programs written in different languages is also present in [143], where PHP programs are linked with flows in embedded languages like SQL, HTML, and JavaScript. Its novelty is how the different language entities are linked in a generic way. In [157] the authors build a dynamic program slicing that benefits from the Microsoft Common Language Runtime (CLR) to be able to deal with all the .NET framework's languages. A relational approach to program slicing is presented in [200]. Their approach is language-independent among the procedural languages thanks to a generalized fact extraction. They define a set of common procedural program entities in order to extract the facts. The EDG is a generalization of this approach because their ASTs are just one particular case over which the EDG can be constructed.

The observation-based slicing (ORBS) [20] is a paradigm-independent slicing technique that iteratively removes lines from a program and checks whether the observable behaviour is the desired one. This is checked for a particular set of test cases. The lines are removed as text, i.e. without the need for any parsing, which makes this approach applicable to any language without any special treatment. However, the quality of the tests directly affects to the quality of the result, in such a way that it can produce incomplete slices if the test cases do not cover key points of the program. Another important problem of ORBS and related approaches such as tree ORBS [21] (which deletes nodes—instead of lines—from the tree representation of the program) is the slicing cost. In many cases, deleting only one line/node results in a syntax error; and several lines/nodes must be deleted to make the slice compilable. The combinatorial explosion of deleting sets of lines/nodes can significantly increase the slicing time due to its exponential cost [152].

Chapter 6

Quasi-Minimal Slicing to Compare Program Slicers

Being able to compute minimal slices would speed up many software processes. For instance, compilers use program slicing to remove dead code, and many analyses use program slicing as a preprocessing stage to detect variable dependences. Therefore, making slicing more accurate would also improve the later analyses based on it.

Because computing minimal static slices is undecidable [204], almost all program slicing techniques guarantee that their computed slices are *complete* (i.e., they contain all statements that do influence the slicing criterion), but, in general, they do not guarantee that their computed slices are *correct* (i.e., they probably contain statements that do not influence the slicing criterion).

Example 6.1. *Consider the following programs:*

```
1  read(z)
2  x = 42;
3  y = 1;
4  x++;
5  if (z > 0) {
6     x--;
7     print(x);
8  }
```

(A) Original Program

```
read(z)
x = 42;
y = 1;
x++;
if (z > 0) {
   x--;
   print(x);
}
```

(B) Slice

```
read(z)
x = 42;
y = 1;
x++;
if (z > 0) {
   x--;
   print(x);
}
```

(C) Minimal Slice

Figure 6.1: Program with a possible slice and a minimal slice w.r.t. $\langle 7, x \rangle$

We can define the slicing criterion $\langle 7, x \rangle$ *in the original program. This means that we are interested in all statements that are needed to compute the value of* x *in line 7. The original code is a slice of itself, but there exist smaller slices. For instance, the code in the middle is the slice computed by almost all current static program slicers (e.g., this is the output of the Indus Java slicer [165] and CodeSurfer [9]). However, the slice in the middle is not minimal. The minimal slice of the original program is the code on the right. It would be difficult for a slicer to compute this slice because line 7 is reachable via a control-flow path from line 6, and line 6 defines variable* x*, which is used in line 7. Thus, most slicers consider that line 6 does influence line 7. This reasoning is transitively applied to lines 4 and 6. Hence, program slicers produce the code in the middle.*

Example 6.1 illustrates how a tiny program without method calls and even without loops, cannot be handled precisely by current program slicers. The fact that computing minimal slices is undecidable in the general case does not, however, prevent us from defining a procedure to compute minimal slices for a given concrete program. Nevertheless, normally, even for very small programs, this procedure would be computationally intractable [20, 212]. Unfortunately, human intervention is often needed to produce minimal slices, and this is only practical for small programs.

In this chapter we propose a method to compute *quasi-minimal slices* (Sections 6.1 and 6.2), which, roughly, are minimal slices for a given set of inputs (this means that quasi-minimal slices may not be sound static slices, i.e., for all possible sets of inputs). In many cases, we are interested in producing a slice with respect to a given computation (known as *minimal dynamic slice*). For instance, in debugging we are often interested in producing a slice of a program that produced an error *for a particular input* because the slice produced is a reduced version of the program that reproduces the wrong computation (and that contains the error). In regression testing, after we test a new release of a program with the regression tests, many different errors can show up. In this situation, we can be interested in producing a slice *for a given set of test cases* (known as *minimal simultaneous dynamic slice*).

Our method produces minimal dynamic slices and simultaneous dynamic slices, and it can also produce static minimal slices in many cases. On the one hand, if the input domain of a program is finite, we automatically produce all possible input values, thus, producing a minimal static slice. On the other hand, if the input domain is infinite, we provide an instrumentation based on concolic testing [127, 176] to produce test cases that explore all possible branches (100% branch coverage) of the program. This ensures in many cases that the produced static slice is also minimal.

From the best of our knowledge, there does not exist any public repository of benchmarks with minimal slices, and this is surprising, because a suite of minimal slices is very useful for slicer developers. In particular, our group have implemented several program slicers for different languages, including Petri nets [125], XQuery [7], Erlang [180], and CSP [124]. Every time a new version of the slicer was prepared, (e.g., with a new technique or feature, or just to correct some bug) the same problem repeated: there is no mechanism to measure the improvement achieved with the performed changes. What the group usually do is to implement some new benchmarks and compare previous results with new ones. This gives a measure of improvement. Contrarily, when a new version is prepared, the ideal scenario would be to start a battery of tests that automatically compare the new slices produced by our released code with a gold standard (i.e., the minimal slices). This is exactly what we have done with Erlang as target language. The elements used to implement the methodology are shown in Section 6.3 and the suite of benchmarks obtained is described in Section 6.4. The quasi-minimal program slices computed would allow us not only to objectively measure the improvement of the new release, but also to detect possible introduced problems in other parts of the slicer, and, e.g., to fairly compare our tool with other tools.

As an application of our method to produce quasi-minimal slices, in this work we have obtained a reliable system to evaluate and compare program slicers. This system inputs a program slicer and outputs a report about precision and recall of this slicer with respect to a suite of minimal slices that have been already computed. The system can also input two slicers and compare them. If the two slicers are two releases of the same slicer, then the system can measure the improvement achieved, but also identify errors introduced (or solved) in the new slicer version.

6.1 Using ASTs to Improve Granularity

Although most program slicers that use dependence graphs use statements as the unit when measuring slicing quality, there are other approaches where program slices are measured in code lines. The reason is that these program slicing techniques consider lines of code as atomic elements and, thus, they remove a whole line or nothing [20, 49, 204]. For this reason, most of the work that compares the precision of different program slicing techniques just compare the retrieved number of lines (see, e.g., [19, 20]). Unfortunately, this is very sensitive to the programming style, and moreover, it can be very imprecise, especially in functional languages.

Example 6.2. *Consider the Erlang program in Figure 6.2a and its minimal slice in Figure 6.2b with respect to the slicing criterion $\langle 10, B\rangle$. Observe that some expressions have been replaced by* `_` *or by the fresh atom* `sliced` *(see [16, 180]). This is needed to make the slice executable. Clearly, all methods based on lines would not be able to remove the subexpressions that are not needed in lines 1, 2, 3, and 6. For instance, in line 2,* `C=B` *can be removed, but the programmer initialized* `A`, `B`, *and* `C` *in a single line, and thus the whole line cannot be removed. One can argue that a preprocessing phase could be used to refactor the code and place all statements in different lines whenever it is possible. But this cannot solve the second problem: sometimes only a subexpression can be removed in a line. This is the case of variables* `Z`, `Y`, *and* `C` *in line 3.*

To overcome these limitations, as already done by, e.g., CodeSurfer [9] or in [180], in our method we propose to use expressions as the slicing criterion so precision can be increased. We also reason about slices at the AST level, so that instead of counting lines of code, we can measure the number of AST nodes that belong to the slices, thus, producing a more precise measure.

In the method proposed in this chapter, besides a program P, a slicing criterion C, and a set of inputs $\mathcal{I}$, we also associate slices with the AST of P. This is particularly useful to allow us to reason about the accuracy of slices. Therefore, we need to adapt the notion of slicing criterion to ASTs. This can be easily done by redefining a slicing criterion in such a way that the point of interest is not an expression, but the AST node whose subtree represents that expression. We define the slicing criterion and the dynamic slicing criterion in terms of ASTs as follows:

```
1   main(X,Y) ->
2     A=1, B=A, C=B,
3     Z=foo(X,{Y,B,C}),
4     Z.
5
6   foo(X,{Y,B,C}) ->
7     case X of
8       123456789 -> Z=X/Y,
9                    Z+C;
10      2 -> B;
11      _ -> X/Y
12    end.
```

(A) Original Program

```
1   main(X,_) ->
2     A=1, B=A,
3     _=foo(X,{sliced,B,sliced}).
4
5
6   foo(X,{_,B,_}) ->
7     case X of
8
9
10      2 -> B
11
12      end.
```

(B) Minimal Slice w.r.t $\langle 10, B\rangle$

FIGURE 6.2: Erlang program and its minimal slice w.r.t $\langle 10, B\rangle$

Definition 6.1 (AST-adapted slicing criterion). *Let P be a program and C a slicing criterion of P. Let $AST(P) = (N, E)$ be an AST of P where N is the set of nodes and E the set of edges. n is the AST-adapted slicing criterion of C such that $n \in N$ and n is the root of the subtree of $AST(P)$ that represents C.*

Definition 6.2 (AST-adapted dynamic slicing criterion). *Let P be a program and $\langle C, I\rangle$ be a dynamic slicing criterion of P. An AST-adapted dynamic slicing criterion of $\langle C, I\rangle$ is a tuple $\langle n, I\rangle$ such that n is the AST-adapted slicing criterion of C.*

6.2 A Method to Produce Quasi-Minimal Slices

Because computing minimal static slices is undecidable, similarly to [20], we can relax its definition to be minimal with respect to a finite set of inputs. Formally,

Definition 6.3 (Quasi-minimal slice). *Let $\mathcal{I}$ be a set of possible inputs for a program P, and C a slicing criterion for P. A* quasi-minimal slice *(QM-slice) qm-slice$(P, C, \mathcal{I})$ of P with respect to C and $\mathcal{I}$ is a dynamic executable program slice of P that is minimal for all $I \in \mathcal{I}$ on a dynamic slicing criterion $\langle C, I\rangle$. If $\mathcal{I}$ contains all possible inputs of P, then qm-slice$(P, C, \mathcal{I})$ is a* minimal slice *of P with respect to C.*

Given a program and a slicing criterion, our method computes its QM-slice following two sequential phases. The first phase produces a static slice of the original program, which is the input of the second phase. It is worth to remark that Phase 1 is optional, If we do not have any static program slicer available to use during this phase it can be omitted. Even so, it is highly recommendable because Phase 1 can significantly reduce the number of AST nodes that Phase 2 has to work with, which speeds up the process (e.g., in our implementation of the method, Phase 1 reduces the time of Phase 2 by 64,99%). The second phase further slices this slice, producing the final QM-slice. Figure 6.3 summarises the method, which is further detailed in the following sections.

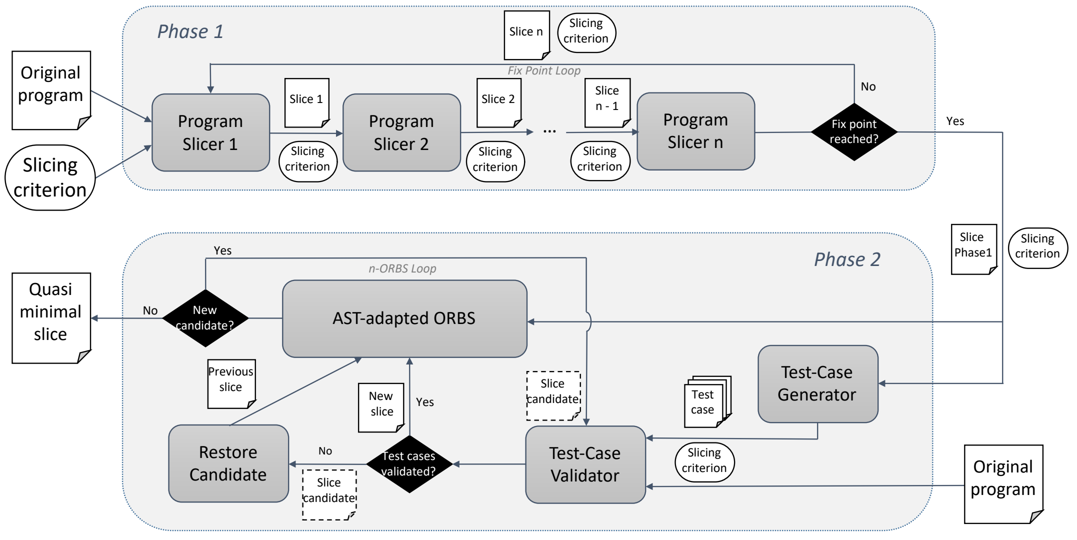


FIGURE 6.3: A method to produce quasi-minimal slices

6.2.1 Phase 1: Combining static program slicers

In the first phase, we use a set of static program slicers to repeatedly slice the original program until a fix point is reached. Different program slicers usually implement different techniques and optimisations to reduce the size of the slice. Therefore, we can use any program slicer to produce a first slice that we can use as the starting point to further reduce its size with another program slicer, because the slice of a slice is a slice provided that the same slicing criterion is used.

Theorem 6.1. *Let P be a program, and $S = slice_{X_1}(P, C)$ be a program slice. Then, $S' = slice_{X_2}(S, C) \in \mathcal{S}lices^P_C$ for any P, C, X_1, and X_2.*

Proof. By point 1 in Definition 2.4, we know that $S' \subseteq S \subseteq P$. By point 2 in Definition 2.4, we know that $\forall I \ : \ seq(P, C, I)$ is a prefix of $seq(S, C, I)$ and that $seq(S, C, I)$ is a prefix of $seq(S', C, I)$. Therefore, $seq(P, C, I)$ is also a prefix of $seq(S', C, I)$. Hence, $S' \in \mathcal{S}lices^P_C$. □

Therefore, given a program P and a slicing criterion C, slicer B can use the slice provided by slicer A as its input and take advantage of the code removed by A. However, A may also take advantage of the code removed by B and, thus, remove code it did not remove the first time, which would imply that A can take further advantage of the new code removed. Therefore, a loop between all the slicers is needed until none of them can further remove any additional code, thus reaching a fix point.

One important property of the slicers, which is a requirement of the method, is that the slices produced by all of them must be complete. Therefore, the output of Phase 1 is always a complete slice, because the sequential composition of complete slicers produces a complete slicer.

Theorem 6.2 (Completeness)**.** *Let P be a program, and let C be a slicing criterion for P. Given two complete program slicers X_1 and X_2, then:*

$slice_{X_2}(slice_{X_1}(P, C), C)$ is a complete slice with respect to P and C.

Proof. First, because X_1 is complete, we know that $S_1 = slice_{X_1}(P, C)$ is a complete slice with respect to P and C. We prove the theorem by contradiction assuming that the slice $S_2 = slice_{X_2}(slice_{X_1}(P, C), C)$ is not complete with respect to P and C. This is only possible if either X_2 is not complete, and thus $slice_{X_2}(S_1, C)$ is not a complete slice with respect to S_1 and C, or if X_2 is complete, but S_1 is not complete with respect to P and C. However, both cases lead to a contradiction because both S_1 and X_2 are complete. Moreover, because S_1 and S_2 are complete, then $S_2 \subseteq S_1 \subseteq P$, and thus, S_2 is also a complete slice with respect to P and C. $\square$

While it is mathematically correct to say that the slicing criterion C is common for all program slicers (because C is a reference to a piece of code in P), in practice C is normally provided in a text mode (e.g., $\langle 5, v\rangle$ meaning line 5, variable v) so it is not a reference anymore. Therefore, if a line before C (e.g., line 2) is sliced off from P by the first program slicer obtaining S, then C needs to be updated (e.g., to $\langle 4, v\rangle$) so the subsequent program slicers can locate the slicing criterion in S. Figure 6.4 shows how the slicing criterion is updated. The process consists of four steps: first, an AST of the code and of its slice are obtained; second, a mapping ([166, 190]) over both ASTs is calculated (dashed lines in the figure); third, the node that represents the slicing criterion (see Definitions 6.1 and 6.2) is located within the AST of the code; finally, the mapping is used to find the node in the AST of the slice.

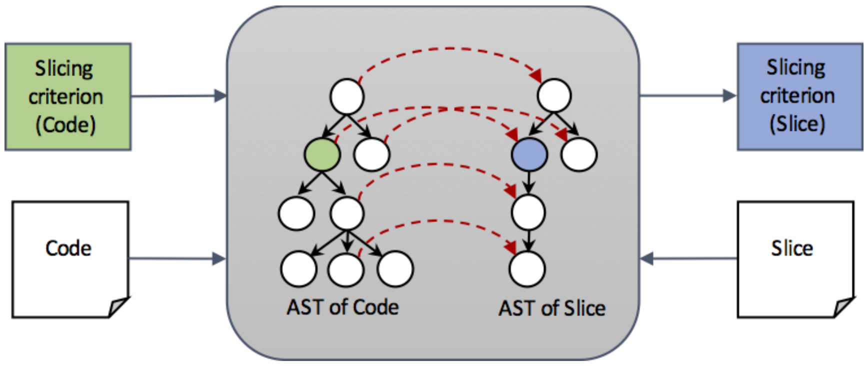


FIGURE 6.4: Slicing criterion mapper

6.2.2 Phase 2: Increasing precision via an AST-adapted ORBS algorithm

Phase 2 consists in an iterative process that continuously reduces the slice until a fix point is reached. It comprises three main modules: ORBS, test-case generation, and test-case validation, which are explained hereafter.

ORBS

We have implemented a variant of observation-based slicing (see Chapter 2, Section 2.4). Our variant of ORBS iterates over the AST of the program (instead of iterating over its lines). In particular, it iterates over the AST of '$Slice_{Phase1}$' (see Figure 6.3). Roughly, this variant iteratively tries to remove from the AST each subtree. Each removal attempt of a subtree produces a '*Slice candidate*' (see Figure 6.3). For each slice candidate, its behaviour is compared with the behaviour of the original program according to Definition 6.3. If they show the same behaviour, then that part is permanently removed from the AST producing a '*New slice*' (see Figure 6.3), and ORBS is restarted with this new slice as input. Otherwise, the '*Previous slice*' is restored and used in a new iteration of ORBS. This iterative process is incremental (first, it removes one node at a time, then, two nodes at a time, and so on) and continues until no more nodes can be removed. This ORBS-based technique is described in Algorithm Algorithm 6.1, where we use E^* to denote the reflexive and transitive closure of E. Note that the algorithm is parametric with respect to MN, which denotes the maximum number of nodes that can be removed to produce a slice candidate.[1] Roughly, the algorithm loops *currMN* from 1 to MN. It proceeds by removing every combination of *currMN* nodes and then testing them. E.g., first take out each one node (and its subtree), run tests to check whether the sequences of values at the slicing criterion are preserved. Second, take out combinations of two nodes (and their subtrees), test those; and so on. Always building on the previous result.

For this, the algorithm uses function SEQ(P, C, t), which executes program P with the test case t and records the sequences of values computed at the slicing criterion C. The recursive function ORBSAST iterates top-down over the AST removing subtrees and checking whether the sequences of values computed with SEQ for the original program are a prefix of the sequences of values computed for the new program with the subtrees removed. This is done with a battery of tests (also called inputs in our context). Function ORBSAST is called until a fix-point is reached (**repeat-until** loop), for each number of removed nodes from 1 to MN (**for** loop).

This adaptation of ORBS to ASTs works top-down. This is more efficient because it works in a concretisation fashion, trying to remove first entire functions, clauses, and data structures before trying with their components. If a bottom-up traversal were used instead, whenever a function could be removed each of its statements would be removed beforehand. This is probably not a problem in other contexts, but in our context each time a subtree (e.g., a statement) is removed from the AST all generated test-cases are run to validate that removal. Clearly, these validations are a waste of time in case the whole function is going to be removed.

The only functions that must be provided by the user in Algorithm 6.1 are SEQ, which computes the sequence of values obtained for a specific execution, and GENERATETESTCASES, which is described in the next subsection.

It is important to remark that our algorithm is a generalisation of ORBS in two ways. First, because it can slice any expression and not only lines of code. If we consider that the removed nodes can only be those subtrees that correspond to lines in the code, then our algorithm is equivalent to ORBS. However, there is a second generalisation. ORBS uses a window of size δ that represent the lines

[1] All possible combinations could be checked when $MN = |N|$.

Algorithm 6.1 ORBS-based AST pruning algorithm

Input: A program P, an executable program slice S of P, a slicing criterion C for S, and the maximum number of nodes MN to be removed at a time.
Output: A quasi-minimal slice of P'.

```
 1: function ComputeQMSlice($P$,$S$,$C$)
 2:   $tests \leftarrow$ generateTestCases($S$,$C$)
 3:   $testsSeq \leftarrow \{t, \textsc{seq}(P, C, t) \mid t \in tests\}$
 4:   $A \leftarrow$ getAST($S$)
 5:   for $currMN \in 1 \ldots MN$ do
 6:     repeat
 7:       $A' \leftarrow A$
 8:       $A \leftarrow$ OrbsAst($A'$, 1, $currMN$, $\emptyset$)
 9:     until $A = A'$
10:   $P' \leftarrow$ getProgram($A$)
11:   return $P'$

12: function OrbsAst($A$, $currNode$, $currMN$, $treatedNodes$)
13:   $\langle N, E \rangle \leftarrow A$
14:   $remNodes \leftarrow \{n \in N \mid \nexists n' \in treatedNodes \,.\, (n', n) \in E^*\}$
15:   while $remNodes \neq \emptyset$ do
16:     $node \leftarrow n \in remNodes \mid \nexists n' \in remNodes \,.\, (n', n) \in E$
17:     $remNodes \leftarrow remNodes \setminus \{node\}$
18:     $N' \leftarrow N \setminus \{n \in N \mid (node, n) \in E^*\}$
19:     $E'_s \leftarrow \{(n, n') \in E \mid n, n' \in N'\}$
20:     $A' \leftarrow (N', E'_s)$
21:     if $currNode < currMN$ then
22:       $A'' \leftarrow$ OrbsAst($A'$, $currNode + 1$, $currMN$, $treatedNodes \cup \{node\}$)
23:       if $A' \neq A''$ then
24:         return $A''$
25:     else
26:       if $\forall \langle t, \textsc{seq}_P \rangle \in testsSeq$ .
                $\textsc{seq}_P$ is a prefix of seq(getProgram($A'$), $C$, $t$) then
27:         return $A'$
28:   return $A$
```

that can be removed all together. Therefore, ORBS can delete various lines at a time, but it imposes the restriction that all of them must be together (inside the window). This means that ORBS cannot produce the minimal slice of Example 6.1, because lines 4 and 6 must be deleted together without deleting line 5. Our approach allows for deleting different (not necessarily adjacent) subtrees of the AST, thus, solving this problem and producing the minimal slice in Example 6.1.

Test-case generation and validation.

The second module used in this phase is in charge of the test-case generation, which is implemented by function *generateTestCases* in Algorithm 6.1. The goal is to generate test cases that execute different paths of the slice and that evaluate the slicing criterion. Every '*Slice candidate*' produced by ORBS is tested by comparing its behaviour with the one of the original program. If they show the same behaviour, then the missing code in the slice candidate is definitely removed. Otherwise, it is restored. Clearly, the quality of this phase depends on the generated test-cases. An important remark is that our architecture takes advantage of Phase 1 not only to produce the refined slice '$Slice_{Phase1}$', but also to improve the generation of test cases. In particular, we can observe in Figure 6.3 that module '*Test-Case Generator*' inputs '$Slice_{Phase1}$' (instead of the '*Original program*'). Generating the test cases from '$Slice_{Phase1}$' produces better test cases because this avoids generating test cases that explore removed code in the slice (and, thus, that cannot affect the slicing criterion). Observe, however, that '$Slice_{Phase1}$' is not used as input for the module '*Test-Case Validator*' because the output of *seq* for this slice and for the '*Original program*' differ according to property 2 in Definition 2.1. This is explained in Example 6.3.

Example 6.3. *Consider the following sequences of values produced in a slicing criterion SC when executing a concrete input I over the original program (*Original*), the output slice of phase 1 (*$Slice_{Phase1}$*) and a slice candidate (*SliceCandidate*).*

a) $seq(Original, I, SC) = [1, 2, 3]$

b) $seq(Slice_{Phase1}, I, SC) = [1, 2, 3, 5]$

c) $seq(SliceCandidate, I, SC) = [1, 2, 3, 7]$

In this scenario, if we validate (i.e., decide whether it is a valid slice) SliceCandidate *with respect to* Original*, then, according to property 2 of Definition 2.4,* SliceCandidate *is an executable program slice of Original ((a) is a prefix of (c)). Nevertheless, if we validate* SliceCandidate *with respect to* $Slice_{Phase1}$*, then the validation fails because (b) is not a prefix of (c). This happens because a slice can produce more values than the original program in the slicing criterion. Therefore, module* Test-Case Validator *inputs* Original *to prevent these kind of false negatives.*

Figure 6.3 summarises the described phases. In the figure: the phases are enclosed inside light grey boxes; the slicers and the other processes are represented with dark grey boxes; the slices and the test cases are represented with white files; the slice candidates (not validated yet or not valid) are represented with dashed-border white files; and decision points are represented with dark rhombuses. The intermediate and output slices of the first phase must be static executable program slices of the original program (see Definition 2.4) whereas the intermediate and output slices of the second phase are dynamic executable program slices (see Definition 2.13).

Note that this is a general scheme that can be adapted to any language. For this, we only need to instantiate some of the dark grey components: the program slicers, the test-case validator, and the test-case generator (the ORBS technique is already paradigm-independent and works for any language).

6.3 Implementation

We have instantiated the method for Erlang. The method follows the schema shown in Figure 6.3, where we use: two program slicers in Phase 1 called *Slicerl* [180] and *e-Knife*; *CutEr* (see Chapter 7, Section 7.1) as a test-case generator; *SecEr* [88] as a test-case validator; and *Cover* [54] as a coverage meter to decide when to stop generating test cases.

6.3.1 Phase 1: *Slicerl* and *e-Knife*

In our setting, we used two slicers: *Slicerl* [180] and *e-Knife*. We selected Slicerl for four reasons: First, because it is based on a data structure called Erlang Dependence Graph (EDG) whose granularity level is minimal (i.e., tokens). This allows for removing expressions even inside a line of code. Second, because it is open-source, and thus, we have been able to access its internal behaviour and analyses, extend it, and use it in our implementation. Third, because it implements some novel optimisation techniques that make it very precise. And fourth, because it is interprocedural. Other slicers such as the Wrangler's slicer [117] were discarded because they are only intraprocedural, and thus they cannot handle with precision any of the benchmarks in the suite.[2]

The second slicer is called e-Knife. It is a static slicer for Erlang on which we have been working for the last few years. e-Knife is also based on the EDG and thus, it has the same granularity level as Slicerl: tokens (every token is represented in the EDG with a different node that is susceptible of being sliced off). Moreover, e-Knife incorporates a new technique to precisely slice composite data structures, which complements the static analyses made by Slicerl.

Example 6.4. *Given the following program on the left, with the slicing criterion $\langle 3, X \rangle$, Slicerl produces the slice in the middle, whereas e-Knife produces the slice on the right:*

```
1  main() ->
2    A = {1,2},
3    {X,Y} = A.
```

Original Program

```
main() ->
  A = {1,2},
  {X,_} = A.
```

Slicerl's Slice

```
main() ->
  A = {1,sliced},
  {X,_} = A.
```

e-Knife's Slice

Note that, even though `X` *depends on* `A`*, and* `A` *depends on* `2`*,* `X` *does not depend on* `2`*. Only e-Knife is able to detect intransitive data dependences.*

[2]Note that this does not mean that the suite is useless for intraprocedural slicers. It just means that intraprocedural slicers are less useful to construct the suite.

6.3.2 Phase 2: CutEr, Cover, and SecEr

For Phase 2, we explain how we have instantiated for Erlang the test-case generation and validation tasks needed for ORBS.

Test-case generation

We ensure high quality test cases using concolic testing. We did two sequential steps to ensure a 100% branch and statement coverage:

- *Concolic test-case generation.* This technique analyses the branching conditions in the source code and generates constraints that the input must satisfy to visit all branches. Then, a constraint solver is used to produce the test cases. We used a concolic testing tool for Erlang called CutEr [68]. The following example shows that white-box testing can generate test cases that execute very unusual branches.

  Example 6.5. *Consider again the program in Figure 6.2a. The case branch in line 8 will be hardly executed with random test-case generation. 100% branch coverage can only be achieved if a test case exists with* `X=123456789`*. However, this does not guarantee the evaluation of all expressions in the branch. 100% statement coverage in the first branch can only be achieved if a test case exists with* `X = 123456789, Y` $\neq$ `0`.

- *Semi-random test-case generation.* We complemented our white-box testing with black-box testing. We implemented random generators for all possible data types in Erlang.

  The maximum number of test cases to be generated is a parameter of our method. This number depends on the concrete code to be processed. The number also has a direct impact on the runtime and on the precision of the final slices produced. In the default configuration, our implementation generates test cases until all the code is tested (i.e., 100% statement and branch coverage). For this, it generates 10 test cases at a time, accumulating the test cases produced and measuring at each step the coverage with a tool called Cover [54]. Cover is a coverage analysis library for Erlang that can determine the coverage achieved when executing a program with several invocations (in our setting, test cases) and that can also identify the uncovered branches. It basically instruments the code so that every line is augmented with a new function call. Therefore, by counting the calls performed during the execution of the test cases we can know exactly what lines were executed and how many times. When Cover reports that 100% statement and branch coverage is reached, the test-case generation finishes. We want to note that these coverages are only metrics, but not objectives. 100% coverage does not necessarily imply high slicing precision.

  Example 6.6. *Consider again the program in Figure 6.2a. A test case with input* `X = 1, Y = 1` *does execute all expressions in line 11—100% branch and statement coverage in this line—, but it does not trigger the* *division-by-zero exception. Finding this situation would require to generate more test cases (e.g.,* `X = 1`*,* `Y= 0`*).*

Test-case validation

Our test-case generation obtains inputs that ensure a 100% statement and branch coverage. However, in our case, these inputs must be complemented with very specific outputs to form the test cases: the sequences of values the slicing criterion is evaluated to. In our case, this is done by another one of our tools called *SecEr* [88], which purpose is further described in Chapter 8 (SecEr implements function SEQ in Algorithm 6.1). Given a slicing criterion, SecEr instruments the source code in such a way that the execution of the instrumented code obtains as a side effect the sequence of values it is evaluated to.

6.4 Experimental Evaluation and Results

We identified a collection of slicing problems and challenges and applied our method to obtain 23 benchmarks for Erlang (23 slicing criteria defined over 18 different Erlang programs) that (combined) implement all of the problems. These benchmarks form a suite that contains triples *program–slicing criterion–minimal slice*. The slices produced in our implementation are QM-slices (see Definition 6.3) and they are fine-grained slices because they have been obtained working over AST nodes (see Definitions 6.1 and 6.2). In this section, we show the behaviour of each component of the method.

6.4.1 Phase 1: Behaviour of Slicerl and e-Knife

The fix point of Phase 1 was reached in only one iteration (the slice produced by e-Knife could not be further reduced by Slicerl). The first slicer (Slicerl) needed 12820 milliseconds to slice all the benchmarks except for nine of them whose syntax was not supported. This produces an average of 916 milliseconds per benchmark. Slicerl was able to remove 619 nodes from '*Original program*' in total (an average reduction of 31.89%). The second slicer needed 48361 milliseconds to slice all the benchmarks (an average of 2103 milliseconds per benchmark)[3]. e-Knife further reduced the slices produced by Slicerl by 59 nodes in total (an average extra reduction of 2.48% over the original program). If we also consider those benchmarks that Slicerl cannot handle, then the extra reduction was 14.67%.

6.4.2 Phase 2: Behaviour of ORBS and CutEr

ORBS

The execution of Algorithm 6.1 with $Slice_{Phase1}$ and removing one node at a time ($MN = 1$) reduced the original program to a 50.08% (as an average).

[3]e-Knife is a slicer implemented in Java. For this reason, it needs extra time to access Erlang.

This is an extra reduction of 15.84% over the result of phase 1. Afterwards, Algorithm 6.1 was executed again but this time removing two nodes instead of one. The slice remained unchanged in all cases (0% reduction). Then, three nodes were removed in each iteration and 0% reduction was achieved again. Finally, four nodes were removed in each iteration for some benchmarks (according to our estimations, the evaluation of the other benchmarks would have taken months). Again, in all cases, 0% reduction was achieved when four nodes were removed in each iteration. Due to the combinatorial explosion, we did not run any of the benchmarks with five nodes, because its run time was estimated in years.

We compare the four iterations performed with ORBS in Table 6.1. The columns labelled with `i nodes`, where $i \in \{1, 2, 3, 4\}$, represent each of the iterations of the **for** loop in Algorithm 6.1 (the first removing 1 node in each iteration, the second removing 2 nodes in each iteration, etc.). In these columns, `Iter` is the number of different iterations performed by the algorithm (i.e., the number of configurations that were checked, where each configuration is the result of removing `i` nodes from the AST), `Time` is the total time used to check the configurations, and `%` is the percentage of nodes that remain from the original code. Note that the algorithm only removed nodes when trying to remove single nodes (`1 node`).

This whole exhaustive process (with $MN = 4$) took nine days, thirteen hours, and fifty one minutes. However, the ORBS loop with 2, 3, and 4 nodes did not produce any reduction (and consumed most of the time). Therefore, unless one is specially interested in producing minimal slices (as we are) it is a good design decision to configure ORBS to only remove one node at a time. This nearly always produces exactly the same results but the time is significantly reduced. With this configuration ($MN = 1$), the whole suite of benchmarks was sliced in 14 minutes and 25 seconds, producing the same results.

Benchmark	**1 Node**			**2 Nodes**			**3 Nodes**			**4 Nodes**		
	Iter	**Time**	**%**	**Iter**	**Time**	**%**	**Iter**	**Time**	**%**	**Iter**	**Time**	**%**
b1_s56Year	10430	441.48s.	28.29%	23155	649.08s.	0%	692795	269172.36s.	0%	-	-	-
b2_s38C	253	10.76s.	26.56%	264	6.08s.	0%	1072	23.00s.	0%	2714	53.34s.	0%
b2_s40D	82	3.51s.	29.69%	467	10.49s.	0%	3460	73.66s.	0%	16110	315.32s.	0%
b3_s28C	133	5.50s.	45.28%	136	3.69s.	0%	395	8.76s.	0%	621	15.35s.	0%
b4_s32Abb	193	8.36s.	72.41%	1453	36.41s.	0%	21319	488.77s.	0%	211141	4437.24s.	0%
b5_s30C	42	2.21s.	94.44%	758	18.64s.	0%	7909	165.45s.	0%	54567	1134.96s.	0%
b6_s35C	222	9.07s.	28.57%	414	10.00s.	0%	2923	61.19s.	0%	13063	257.84s.	0%
b6_s36D	126	5.33s.	20.30%	206	4.82s.	0%	836	17.90s.	0%	2039	39.64s.	0%
b7_s27C	18	0.87s.	78.57%	105	3.38s.	0%	234	5.33s.	0%	285	7.54s.	0%
b8_s29Deposits	244	23.14s.	72.50%	1165	51.24s.	0%	15468	540.83s.	0%	140135	4134.79s.	0%
b9_s59A	112	4.96s.	88.14%	799	18.83s.	0%	8039	177.06s.	0%	51658	1353.71s.	0%
b10_s34DB	253	43.48s.	76.43%	4383	225.84s.	0%	123884	4651.84s.	0%	2477094	81536.00s.	0%
b11_s28C	28	1.18s.	80.00%	293	7.00s.	0%	1588	44.10s.	0%	5509	133.16s.	0%
b12_s40BS	138	12.56s.	25.77%	4872	257.64s.	0%	145830	5510.94s.	0%	3085914	102958.79s.	0%
b12_s92A	83	7.17s.	20.70%	3168	166.99s.	0%	74649	2625.25s.	0%	1227042	37374.00s.	0%
b13_s38NewI	41	0.65s.	69.70%	700	29.19s.	0%	6649	208.36s.	0%	39929	1112.95s.	0%
b14_s44V	214	6.78s.	26.32%	983	22.33s.	0%	11418	224.08s.	0%	84958	1689.93s.	0%
b14_s45W	137	156.76s.	23.92%	808	465.78s.	0%	8169	4563.61s.	0%	54519	30204.37s.	0%
b14_s46Z	127	5.35s.	18.18%	357	8.16s.	0%	2402	47.41s.	0%	10629	215.53s.	0%
b15_s65Shown	657	34.97s.	81.33%	13208	384.85s.	0%	661963	16267.98s.	0%	23716885	560087.23s.	0%
b16_s58C	25	1.37s.	27.78%	241	13.07s.	0%	1195	75.92s.	0%	3465	399.11s.	0%
b17_s54X	408	38.39s.	68.35%	848	69.63s.	0%	9230	3263.62s.	0%	-	-	-
b18_s50J	341	41.25s.	48.42%	588	50.01s.	0%	4965	3761.92s.	0%	-	-	-
AVG	**622.04**	**37.61s.**	**50.08%**	**2581.35**	**109.27s.**	**0%**	**78538.78**	**13564.32s.**	**0%**	**1356446.83**	**35976.58s.**	**0%**
MEDIAN	**137**	**7.17s.**	**45.28%**	**758**	**22.33s.**	**0%**	**7909**	**208.36s.**	**0%**	**45793.50**	**1123.95s.**	**0%**
TOTAL	**14307**	**865.09s.**	**1151.67%**	**59371**	**2513.15s.**	**0%**	**1806392**	**311979.34s.**	**0%**	**31198277**	**798490.53s.**	**0%**

TABLE 6.1: Comparison of the different iterations of ORBS

CutEr

The coverage achieved by the test cases generated with CutEr for each benchmark is listed in column `CutEr` of Table 6.2. In 14 out of 23 benchmarks CutEr produced a 100% branch coverage. In 4 out of 23 benchmarks CutEr produced a branch coverage <100%. In 5 benchmarks (*b16_s58C, b12_s40BS, b12_s92A, b15_s65Shown, b18_s50J*), CutEr returned an error or was unable to generate any test case.

In all those benchmarks where CutEr did not produce a 100% branch and statement coverage, a second phase of semi-random test-case generation was activated to reach 100%. Column `Random` of Table 6.2 shows this second phase where a 100% statement and branch coverage was achieved in only 0.12 seconds on average.

6.4.3 Empirical evaluation

Prior to the design and application of our method, we first produced the slices of the benchmarks with slicerl and with e-Knife, separately. This enables evaluating how precise QM-slices (obtained with our method) are compared to standard slices (obtained with two program slicers).

Executable program slices

We sliced all the benchmarks with two Erlang program slicers (Slicerl and *e-Knife*) that produced an interesting result: the empirical evaluation of (and a comparison between) each slicer. Slicerl could not handle nine of the benchmarks (it crashed due to unhandled syntax constructs). If we omit these benchmarks, then their precision was similar. As an average, Slicerl reduced the original programs ($\overline{X} = 31.90\%, \sigma = 21.29\%$), while *e-Knife* reduced them ($\overline{X} = 33.03\%, \sigma = 25.18\%$). However, because the analyses performed by both slicers are different, Slicerl was better twice and *e-Knife* was better thirteen times. This clearly justifies the combination of program slicers in the first phase of our method. We also compared the following three slices for all benchmarks:

$$S_1 = slice_{Slicerl}(slice_{e-Knife}(B, C), C)$$
$$S_2 = slice_{e-Knife}(slice_{Slicerl}(B, C), C)$$
$$S_3 = slice_{Slicerl}(B, C) \cap slice_{e-Knife}(B, C)$$

where B is a benchmark and C is a slicing criterion.[4] We discovered that, for all benchmarks, $S_1 = S_2 \subseteq S_3$. Hence, (i) the order in which the slicers were executed was not relevant, and (ii) it is better composing slicers sequentially (i.e., slicing slices) than composing them in parallel and get the intersection. The reason is that one slicer can take advantage of the parts removed by the other slicer. This justifies the need for a fix-point loop in Phase 1 of the method.

[4]Note that, theoretically, unions and intersections of slices are not necessarily slices [48], but in practice (e.g., with all our benchmarks), they usually are.

Quasi-minimal slices

Table 6.2 summarises the empirical evaluation of our particular implementation of the proposed method. Concretely, it compares the size of the successive refinements of all the slices, and the time needed by all processes of the two phases. Each row represents a different benchmark. For each benchmark, column `Nodes` represents its number of AST nodes, which corresponds to the size of the programs/slices. In the case of the slices, we also include the percentage of nodes that remain in the slice with respect to the original program. Column `Time` shows the time expended in each phase measured in seconds (s). Phase 2 is divided into two different processes: test-case generation and ORBS limited to only one iteration. Finally, column `Iterations` shows the number of configurations checked by ORBS (that is, the number of different nodes removed to produce slice candidates).

It is important to compare the data of the different rows taking into account that the columns provide complementary information. For instance, if we compare the reduction % achieved by *Slicerl* for benchmarks b5_s30C and b14_s44V, one can think that the slice produced for b14_s44V is much better (it was reduced to 37.8%, while b5_s30C was only reduced to 94.44%). However, if we observe the % in the ORBS column, we can see that the conclusion could be the opposite: *Slicerl* produced a minimal slice for b5_s30C, while the slice produced for b14_s44V was not minimal.

Evaluation conclusions

Our implementation of the proposed method and its empirical evaluation have answered several research questions in the process:

1. **Is Phase 1 really needed?** The final slice produced in Phase 2 is the same with independence of whether Phase 1 is used or not. However, the use of Phase 1 reduced the time of Phase 2 by 64,99%.

2. **Runtime: How long does each process last?** The whole suite was sliced in 1054 s. (Phase 1: 62 s., ORBS: 865 s., test case generation: 127 s.). This provides an idea of the relative costs of the phases.

3. **Accuracy: How accurate is each phase (on average)?** Phase 1 reduced the original program 34.07%, Phase 2 further reduced it 15.85% (producing the minimal slice).

4. **Concolic vs. random test cases: Is concolic testing enough?** No. Cuter was able to produce the desired coverage 60.87% of the times. In the other 39.13%, random test case generation was needed.

5. ***Slicerl* vs. *e-Knife*: Which is better (on average)?** When they were run independently, *e-Knife* was better in 13/23 benchmarks. Table 6.3 shows the comparison of both slicers.

6. **Sequential vs. parallel composition (intersection) of slicers: Which is better?** Sequential composition of slicers provides the best results.

	Original	Phase 1				Phase 2					Total
		Slicerl		e-Knife		Test-Case Generation		ORBS (one node at a time)			
	Nodes	Time (s)	Nodes	Time (s)	Nodes	CutEr	Random	Iterations	Time (s)	Nodes	Time (s)
b1_s56Year	820	-	-	3.21	649 (79.15%)	100% (4.93s)	–	10430	441.48	181 (28.54%)	449.62
b2_s38C	128	0.87	95 (74.22%)	2.22	95 (74.22%)	100% (16.03s)	–	253	10.76	34 (26.56%)	29.88
b2_s40D	128	0.88	98 (76.56%)	2.32	98 (76.56%)	100% (16.01s)	–	82	3.51	38 (29.69%)	22.72
b3_s28C	53	0.82	35 (66.04%)	2.12	35 (66.04%)	100% (3.15s)	–	133	5.50	24 (45.28%)	11.59
b4_s32Abb	87	-	-	2.19	74 (85.06%)	100% (10.45s)	–	193	8.36	63 (72.41%)	21.00
b5_s30C	54	0.99	51 (94.44%)	2.17	51 (94.44%)	78% (5.03s)	100% (0.02s)	42	2.21	51 (94.44%)	10.42
b6_s35C	133	1.02	62 (46.62%)	2.18	57 (42.86%)	100% (10.87s)	–	222	9.07	38 (28.57%)	23.14
b6_s36D	133	1.01	49 (36.84%)	2.16	49 (36.84%)	100% (3.29s)	–	126	5.33	27 (20.30%)	11.80
b7_s27C	28	0.95	22 (78.57%)	2.12	22 (78.57%)	100% (2.61s)	–	18	0.87	22 (78.57%)	6.55
b8_s29Deposits	80	0.95	78 (97.50%)	2.23	76 (95.00%)	86% (12.69s)	100% (0.03s)	244	23.14	58 (72.50%)	39.03
b9_s59A	59	0.81	57 (96.61%)	2.18	54 (91.53%)	100% (14.07s)	–	112	4.96	52 (88.14%)	22.02
b10_s34DB	140	-	-	2.25	111 (79.29%)	82% (4.96s)	100% (0.02s)	253	43.48	107 (76.43%)	50.71
b11_s28C	40	0.83	32 (80.00%)	2.15	32 (80.00%)	100% (3.96s)	–	28	1.18	32 (80.00%)	8.12
b12_s40BS	454	-	-	2.52	118 (25.99%)	–	100% (0.03s)	138	12.56	117 (25.77%)	15.11
b12_s92A	454	-	-	2.18	94 (20.70%)	–	100% (0.03s)	83	7.17	94 (20.70%)	9.38
b13_s38NewI	66	0.94	46 (69.70%)	2.15	46 (69.70%)	100% (1.65s)	–	41	0.65	46 (69.70%)	5.38
b14_s44V	209	0.93	79 (37.80%)	2.18	75 (35.89%)	100% (1.69s)	–	214	6.78	55 (26.32%)	11.58
b14_s45W	209	0.96	89 (42.58%)	2.20	64 (30.62%)	67% (2.99s)	100% (0.83s)	137	156.76	49 (23.92%)	163.74
b14_s46Z	209	0.86	117 (55.98%)	2.23	97 (46.41%)	100% (4.54s)	–	127	5.35	38 (18.18%)	12.98
b15_s65Shown	225	-	-	2.40	195 (86.67%)	–	100% (0.04s)	657	34.97	183 (81.33%)	37.41
b16_s58C	108	-	-	1.67	30 (27.78%)	–	100% (0.04s)	25	1.37	30 (27.78%)	3.09
b17_s54X	79	-	-	1.01	76 (96.20%)	100% (7.38s)	–	408	38.39	54 (68.35%)	46.79
b18_s50J	95	-	-	1.02	92 (96.84%)	–	100% (0.04s)	341	41.25	46 (48.42%)	42.32
AVERAGE	173.52	0.56	146.61 (80.59%)	2.13	99.57 (65.93%)	95.17% (5.49s)	100% (0.12s)	622.04	37.61	64.91 (50.08%)	84.61
MEDIAN	128	0.93	87 (94.44%)	2.18	75 (76.56%)	100% (4.95s.)	100% (0.03s.)	137	7.17	24 (45.28%)	22.37
TOTAL	3991	12.82	3372 (80.59%)	49.08	2290 (65.93%)	(126.30)	(1.08s)	14307	865.09	1493 (50.08%)	2199.86

TABLE 6.2: Empirical evaluation of the method instantiated for Erlang

	Original	Slicerl		e-Knife	
	Nodes	Time (s)	Nodes	Time (s)	Nodes
b1__s56Year	820	-	-	3.21	647 (78.90%)
b2__s38C	128	0.87	95 (74.22%)	2.07	95 (74.22%)
b2__s40D	128	0.88	98 (76.56%)	2.08	98 (76.56%)
b3__s28C	53	0.82	35 (66.04%)	2.08	35 (66.04%)
b4__s32Abb	87	-	-	2.19	74 (85.06%)
b5__s30C	54	0.99	51 (94.44%)	2.09	51 (94.44%)
b6__s35C	133	1.02	62 (46.62%)	2.10	72 (54.14%)
b6__s36D	133	1.01	49 (36.84%)	2.09	49 (36.84%)
b7__s27C	28	0.95	22 (78.57%)	2.16	22 (78.57%)
b8__s29Deposits	80	0.95	78 (97.50%)	2.08	76 (95.00%)
b9__s59A	59	0.81	57 (96.61%)	2.06	54 (91.53%)
b10__s34DB	142	-	-	2.78	113 (79.58%)
b11__s28C	40	0.83	32 (80.00%)	2.09	32 (80.00%)
b12__s40BS	454	-	-	2.52	118 (25.99%)
b12__s92A	454	-	-	2.18	94 (20.70%)
b13__s38NewI	66	0.94	46 (69.70%)	2.17	46 (69.70%)
b14__s44V	209	0.93	79 (37.80%)	2.27	99 (47.37%)
b14__s45W	209	0.96	89 (42.58%)	2.28	64 (30.62%)
b14__s46Z	209	0.86	117 (55.98%)	2.28	97 (46.41%)
b15__s65Shown	225	-	-	2.39	195 (86.67%)
b16__s58C	108	-	-	1.67	30 (27.78%)
b17__s54X	79	-	-	1.01	76 (96.20%)
b18__s50J	95	-	-	1.02	92 (96.84%)
AVG	173.52	0.92	65 (68.10%)	2.12	101.26 (66.97%)
AVGTotal	173.52	0.56	146.61 (80.59%)	2.12	101.26 (66.97%)

TABLE 6.3: Empirical evaluation and comparison of Slicerl and e-Knife

6.4.4 A suite of minimal slices

Following the method presented, we have generated a suite of minimal slices for Erlang. In Erlang, this suite is especially useful because it presents special challenges for program slicing (higher order, anonymous functions, pattern matching, etc.) and, moreover, in this language no studies evaluating current program slicers existed yet.

Selection of benchmarks

The suite of benchmarks has been designed to contain small to medium programs that contain well-known program slicing challenging problems described in the literature (e.g., dead code, unreachable clauses [175], pattern matching [180], collapse and expansion of composite data structures [198], etc.). For instance, the suite includes classical slicing programs used in different papers such as word count, the SCAM mug, the Montréal boat example, the Horwitz et al. interprocedural slice [84], etc. The objective is to challenge program slicers to check how many of these programs are they able to slice. In order to test different syntax constructs in Erlang that are also challenging for program slicing (e.g., list comprehensions, block structures, chars, remote function calls, etc.), various benchmarks have been taken from the *github* repository and the *rosetta* *code* programming chrestomathy website[5]. For each benchmark, we defined different slicing criteria so that their slices can be used to test slicers that work at the function, clause, line, or expression level. The suite of benchmarks has been designed to contain small to medium programs that:

- Require interprocedural techniques. Interprocedural slicing is a challenge in functional languages. For instance, the program slicer of Wrangler [116], one of the most advanced Erlang refactoring tools, is still intraprocedural.
- Can be sliced by slicers of different performance. The main goal of the suite is not performance, but precision. Therefore, we prefer small to medium programs for which we can systematically produce minimal slices rather than large programs for which reasoning about minimality is impossible due to its prohibitive cost.
- Contain different slicing problems. In fact, each benchmark defines concisely one specific slicing challenge.

This suite can be used to evaluate and compare program slicers, but it is also particularly useful to develop slicers. To help in this last task, we have implemented a tool that inputs a program slicer and it slices all the benchmarks in the suite with this program slicer. Then, the slices the program slicer obtains are compared with the minimal slices in the suite to calculate the accuracy in terms of preserved AST nodes (i.e., using the minimum granularity). Finally, a report indicating the recall, precision, and F1 is provided to the user as well as the variation of these metrics with respect to the best results the program slicer has achieved so far. The suite and the tool are publicly available at:

`https://mist.dsic.upv.es/bencher/`

Structure of the suite

All benchmarks are labelled so that their purposes and properties can be identified by just looking at their labels. The labels classify the benchmarks depending on the slicing challenges they include, and on the syntax constructs they use.

Example 6.7. *All benchmarks are identified with a code. For instance, benchmark b15_s65Shown refers to program 15 with slicing criterion* $\langle 65, Shown \rangle$. *The code of program 15 was originally extracted from* rosettacode. *Then, the code was augmented and redesigned to include challenging problems for slicing. Finally, this benchmark has been labelled with: IP, LC, AF, Rem. Their meanings are:*

IP: *The benchmark requires interprocedural slicing.*

LC: *The benchmark uses* `list comprehensions`.

AF: *The benchmark defines and uses anonymous functions.*

[5]http://rosettacode.org/

Rem: *The benchmark contains remote procedure calls to external functions (non-available code).*

All the information about the meaning of the labels and about the classification of benchmarks can be found on the public website of the suite.

Minimality

Our method/slicer produces quasi-minimal slices. Ensuring minimality is undecidable because not all possible test cases can be executed (they are potentially infinite). However, our method palliates this problem with a test-case generation phase that ensures 100% branch and statement coverage combining white and black box testing. Thanks to this phase, the quasi-minimal slices produced are actually minimal in many cases. In particular, we have manually proved that all 23 quasi-minimal slices generated with our tool (with MN=1 and generating random test cases until 100% statement and branch coverage is achieved) are in fact minimal slices. Concretely, we have proven minimality for each single pair ⟨benchmark, slicing criterion⟩ in the suite, proving that each single node of the sliced AST is actually needed, and that all required nodes are part of the slice. Each benchmark of the suite is, thus, accompanied with a proof of minimality.

6.5 Related Work

One approach similar to ours is dynamic program dicing, proposed by Chen and Cheung [34] as an alternative to static program dicing, which was originally proposed by Lyle and Weiser [130]. This approach obtains a program slice formed by the statements contained in the traces of a set of failed executions and not in the traces of a set of correct executions. In most cases, the remaining statements would contain the source of the error. Nevertheless, this approach presents two differences with respect to our approach: 1) It is incomplete. The slices produced may not contain the errors that produced the discrepancies. (2) The slices produced may not be executable, thus, they cannot be used to check the discrepancies.

In our approach, we use a technique that can be considered a variant of ORBS [20]. ORBS is a language-independent technique, thus it removes lines without parsing them. Hence, if two statements are placed in the same line, they are removed together. Of course, this can also produce compilation errors if a part of one syntax construct is removed. Instead of removing lines of code, we use a mechanism to remove expressions or replace them by a fresh constant `sliced`, thus, the obtained precision is higher and, moreover, this enables us to remove expressions with independence of how they were coded. Our proposed ORBS algorithm is similar to the one proposed in [21] that removes nodes one by one. Specifically, the algorithm proposed in this work is a generalization of [21], because we also allow to iteratively remove N nodes (instead of one) by computing all possible combinations with an efficient top-down pruning algorithm.

Our technique is also similar to Delta Debugging (DD) [42, 212]. DD was originally defined for debugging, but it can also be used to compute slices. The way in which DD and our technique compute slices differ. DD relies on the use of a trace, which is cut in the middle first, in a quarter next, and so on. This process is too expensive compared to our approach (and also compared to ORBS). Moreover, DD can produce slices that are not correct, in the sense that their behaviour differs from the one of the original program. Clearly, this is useless for our purposes, because we need to ensure that the slices of the suite are correct.

Another related approach is Critical Slicing [49]. The idea behind Critical Slicing is the same as ORBS: they both remove lines and check whether the slice produced removing each line preserves the original behaviour. The difference is that Critical Slicing removes lines one at a time, while ORBS removes them incrementally. As a consequence, (i) contrarily to ORBS, Critical Slicing needs a fixed number of compilations (one per line), and (ii) critical slices can be incorrect because two lines individually removed without changing the behaviour at the slicing criterion may produce a program with a different behaviour when they are removed together. Hence, as DD, Critical Slicing also produces incorrect slices.

Comparing and evaluating the performance of program slicers and slicing-based techniques has been traditionally of wide interest. Not only because this enables developers to select the best slicer or technique for their purposes, but also because it provides information about how precise the slicer is. For this reason, many surveys and works exist (see, e.g., [19, 69, 81]) that evaluate and compare the size of the slices produced by different techniques. Unfortunately, due to the lack of a standard suite of benchmarks, in most cases the benchmarks are implemented from scratch to make the experiments [69, 81], they are taken from different papers and projects [20], or they belong to suites of programs not specific for slicing [19]. Moreover, often, the benchmarks that are used in the experiments are not publicly available or accessible (e.g., in [19, 69, 81]), which makes it impossible to replicate and/or validate the study. Furthermore, the unavailability of the benchmarks prevents other researchers and developers from comparing their techniques with the reported results. In consequence, these reports are just a fixed picture of the state of the art, but they are not usable to measure and compare future techniques.

A suite of program slicing benchmarks would solve these problems, but we are not aware of any suite of benchmarks prepared for slicing, i.e., with specific challenging problems for slicing, and with solutions (minimal slices) for each benchmark. The construction of this suite is completely novel. Unfortunately, computing the minimal slices of each benchmark is not trivial at all. In fact, it is undecidable in the general case, so we had to manually prove minimality. The techniques used in this system are very related to other existing techniques and methods. In particular, we use semi-random test-case generation similar to the one implemented by SmallCheck [172]. We also prevent duplicated test cases but, contrarily to SmallCheck, our test-case generation is not based on properties.

Part III

Testing & Verification

Chapter 7

Preliminary Definitions and Notation

This chapter presents a set of testing and verification concepts, together with some tools used along this part of the thesis. The aim of this chapter is to provide the user an overall idea of the purpose of each tool and technique explained, to ease the understanding of the following chapters.

7.1 Analysis tools for Erlang

Most of the techniques explained in this part of the thesis are mainly developed for the Erlang programming language. For this reason, we used different analysis tools already developed for Erlang as components of our approaches. Here, we show an overall idea about some Erlang analysis tools, briefly describing the kind of analysis they perform in Erlang.

7.1.1 Type inference in Erlang: TypEr

Erlang is a strict, dynamically typed functional programming language that supports communication, concurrency, fault-tolerance and distribution. Erlang contains a set of basic data types: atoms, numbers (integers and floats), process identifiers, and compound data types that are represented by lists and tuples. Additionally, a notation for structured objects is also supported (called records in Erlang), but their inner representation in Erlang is currently the same as the one used for tuples. In Erlang, functions are defined as a set of ordered clauses, which may contain associated guards, and the clause selection process is done by pattern matching. Despite the lack of static types, explicit pattern matching against clauses with terms or the presence of guards, generally provides information about the function input and return types.

TypEr [122] is a fully automatic type annotator for Erlang programs based on constraint-based type inference of success typings (a type signature that over-approximates the set of types for which a function can evaluate to a value [121]). Given an Erlang function, TypEr is able to automatically compute the sets of possible input and output types of this function. TypEr's type annotations are always conservative over-approximations, i.e., the type annotations produced by TypEr for a function's arguments may be more general that the real types accepted in practice.

For example, consider the following Erlang function with two different clauses that computes the length of a list.

```
length([]) -> 0;
length([_|T]) -> 1 + length(T).
```

When we run TypEr to obtain the function possible types we obtain the following type specification for function `length`:

```
-spec length([any()]) -> non_neg_integer().
```

The specification given by TypEr indicates that the argument must be a *list* with elements of *any* type and the function result will always be a non-negative integer.

7.1.2 Property-based testing in Erlang: PropEr

Nowadays there are lots of Erlang modules or even complete applications where each function defines its own type specification (spec). Type declarations and function specifications are not used to guarantee type safety, because the Erlang language already provides this guarantee through runtime checks. Instead, they provide a very valuable information about every function, that can be used for testing purposes, e.g., property-based testing.

PropEr [151] is an open-source QuickCheck-inspired property-based testing tool for Erlang. The tool can turn function specifications into simple properties and test the correspondence between the programmers' intentions and the implementation of their functions. For this purpose, it uses types, both module-local and remote, to generate values of each given type, transforming the static spec into a set of particular calls that test the type properties defined in the function signature. PropEr provides an automatic test of functions exclusively based on their type signature divided in three phases:

(i) generation of valid inputs,

(ii) execution of the function with the generated inputs, and

(iii) check of the correspondence between the actual returned value and the output expected type.

7.1.3 Concolic testing in Erlang: CutEr

Concolic testing [70, 113, 176] is a testing method for input generation where a program is simultaneously executed both concretely and symbolically to maximise the achieved path coverage. In concolic testing, test inputs are generated from the execution of the program. The main idea of this testing approach is to collect, during runtime, symbolic constraints that the program establishes over certain input values. This way, by solving these sets of symbolic constraints, we can infer new input values to force the execution of the program to follow a specific path.

CutEr [68] is a tool that applies the idea of concolic testing to detect bugs in Erlang applications, especially bugs that are very difficult to find using other methods. For that purpose, CutEr tries to explore all execution paths to find those that lead to runtime execution errors. The input of CutEr is a program and a particular input for this program. CutEr executes this input running the program concretely and symbolically, and accumulates all the symbolic constraints it finds during the execution. All the collected symbolic constraints are later used to generate new input values by solving them with a particular constraint solver (the Z3 SMT-solver). This process is repeated with the new program inputs computed, accumulating new sets of symbolic constraints that appear during each new execution. The process finishes when there is no more constraint sets to solve and, thus, no more execution paths can be produced.

7.2 Design by contract

The term design-by-contract (DBC) was originally proposed by Bertrand Meyer [134] to formally define the requirements of software systems. The main idea is that a module inside a system grants some kind of contract, or obligation, for the functionality it provides. Consequently, using the module implies the acceptance of the terms of that contract. The specification of this contract is built upon Hoare logic, a formal system developed by Tony Hoare for reasoning about the correctness of programs [80] that allows for the use of composite predicates to describe complex properties. The fundamental basis of this model are threefold:

1. **Preconditions**. They represent requirements that must be met before calling a function. If the preconditions are not fulfilled, then the associated function may compute erroneous values or crash during the execution.

2. **Postconditions**. They represent conditions that must be met after the execution of the function call. The violation of a postcondition implies that a function shows a wrong behaviour during its execution and does not behave as expected.

3. **Invariants**. They are constraints that must be satisfied during the lifetime of a certain component. The infringement of an invariant defined over a specific component may lead to scenarios not considered by the component, resulting into the outbreak of unexpected program results.

The best way to specify the details of a contract is to include it in the module documentation of the API. Functions should specify any expected precondition that must be fulfilled before executing a function call as well as any resulting postconditions that must be satisfied after the function call is completed. Additionally, the documentation should also specify any invariants that must remain true during the observable lifetime of each program component, that is, after construction, before destruction, and before and after each function call.

To show the concept of DBC more concretely, we provide the following example:

Example 7.1. *Consider the Java program with the class* ***String*** *in Figure 7.1. There are two different types of contracts defined as comments in the code. The first one is an invariant stablished over the objects of class* ***String*** *in Line 2, which represents that the size of a String (call to method* ***size()****) must always be greater or equal than zero. On the other hand, we also have a postcondition in Line 10 associated to method* ***append****. The postcondition exposes that the length of the string object being changed must grow by the length of the input string* ***s*** *(the term* ***@pre.size()*** *is used to represent the length of the string before the method is called).*

```
 1 // A container that operates on sequences of characters.
 2 // Invariant: size() >= 0
 3
 4 class String {
 5   ...
 6   // Returns the number of characters in this String object
 7   public int size(){ ... }
 8   ...
 9   // Append 's' to this String object.
10   // Postcondition: size() == @pre.size() + s.size()
11   public void append(String s){
12     ...
13   }
14   ...
15 }
```

FIGURE 7.1: Java program that makes use contracts of DBC approach

Since it is directly related with the code of the module being tested, the DBC approach is often considered as unit testing too. In fact, contracts and the information they provide are used in testing in different ways. Some of these ways are the use of contracts to built runtime assertions checked during the execution of the program, or the so called contract-based testing, introduced by Aichernig [4], that generates tests for the program from the information provided by contracts.

Chapter 8

Software Evolution Control in Erlang

During its useful lifetime, a program might evolve many times. Each evolution is often composed of several changes that produce a new release of the software. There are multiple ways to control that these changes do not modify the behaviour of any part of the program that was already correct. Most of the companies rely on *regression testing* [162, 210] to assure that a desired behaviour of the original program is kept in the new version, but there exist other alternatives such as the static inference of the impact of changes [95, 115, 137, 184].

Even when a program is perfectly working and it fulfils all its functional requirements, sometimes we still need to improve parts of it. There are several reasons why a released program needs to be modified. For instance, improving the maintainability or efficiency; or for other reasons such as obfuscation, security improvement, parallelisation, distribution, platform changes, and hardware changes, among others. In the context of software maintenance it is common to change an algorithm several times until certain performance requirements are met. Often, during each iteration, the code naturally becomes more and more complex and, thus, more difficult to understand and debug. Although regression testing should be ideally done after each change, in real projects the methodology is really different. As reported in [50], only 10% of the companies do regression testing daily. This means that, when an error is detected, it can be hidden after a large number of subsequent changes. The authors also claim that this long-term regression testing is mainly due to the lack of time and resources.

Programmers that want to check whether the semantics of the original program remains unchanged in the new version usually create a test suite. There are several tools that can help in all this process. For instance, Travis CI can be easily integrated in a GitHub repository so that each time a pull request is performed, the test suite is launched. We present here an alternative and complementary approach that creates an automatic test suite to do regression testing: (i) An alternative approach because it can work as a standalone program without the need of using other techniques. Therefore, our technique can check the evolution of the code even if no test suite has been defined. (ii) A complementary approach because it can also be used to complement other techniques, providing a major reliability in the assurance of behaviour preservation.

More sophisticated techniques, but with similar purpose, have been recently announced like the Ubisoft's system [207] that is able to predict programmer errors beforehand. It is quite illustrative that a game-developer company was the first one in presenting a project like this one. The complex algorithms used to simulate physical environments and AI behaviours need several iterations in order to improve their performance. It is in one of those iterations where some regression faults can be introduced.

In the context of debugging, programmers often use breakpoints to observe the values of an expression during an execution. Unfortunately, this feature is not currently available in testing, even though it would be useful to easily focus the test cases on one specific point without modifying the source code (as it happens when using asserts) or adding more code (as it happens in unit testing). This work introduces the ability to specify *points of interest* (POI) in the context of testing. A POI can be any expression in the code (e.g., a function call) meaning that we want to check the behaviour of that expression (the sequence of values it is evaluated to during the execution). Although they handle similar concepts, our POIs are not exactly like breakpoints, since their purpose is different. Breakpoints are used to indicate where the computation should stop, so the user can inspect variable values or control statements. In contrast, a POI defines an expression whose sequence of evaluations to values must be recorded, so that we can check the behaviour preservation (by value comparison) after the execution. In particular, note that placing a breakpoint inside a unit test is not the same as placing a POI inside it because the goals are different.

In our technique, (1) the programmer identifies a POI and a set of *input functions* whose invocations should evaluate the POI. Then, by using a combination of random test case generation, mutation testing, and concolic testing, (2) the tool automatically generates a test suite that tries to cover all possible paths that reach the POI (trying also to produce execution paths that evaluate the POI several times). In our setting, each test case is divided in two different parts: the *input of a test case* (ITC), which is defined as a tuple $\langle f, a \rangle$ where f is the input function to be called in the test case and a a set of specific arguments for the call, and the output, which is the sequence of values the POI is evaluated to during the execution of the ITC. For the sake of disambiguation, in the rest of the chapter we use the term *traces* to refer to these sequences of values. Next, (3) the test suite is used to automatically check whether any *mismatching trace* (also referred as *error* or *unexpected behaviour* during this chapter) is found across both code versions. This is done by passing each individual test case (which contains calls to the input functions) against the new version, and checking whether the same traces are produced at the POI. Finally, (4) the user is provided with a report about the success or failure of these test cases. Note that, as it is common in regression testing, this approach only works for deterministic executions. However, this does not mean that it cannot be used in a program with concurrency or other sources of non-determinism, it only depends on where the POIs are placed and the input functions used (this is further explained in Section 8.5).

We have implemented the approach in a tool named SecEr (*Software Evolution Control for Erlang*), which is publicly available at: `https://github.com/mistupv/secer`. Instead of reinventing the wheel, some of the analyses performed by our tool are done by other existing tools previously described in Chapter 7, Section 7.1: TypEr, PropEr, and CutEr. However, all the analyses performed by SecEr are completely transparent to the user.

Example 8.1. *In order to show the idea behind our approach, we provide an example to compare two versions of a simple Erlang program which has been well and erroneously refactored. All the versions of this program can be seen in Figure 8.1.*

```
main(X,Y) ->
  A = X + Y,
  D = X - Y,
  A * D.
```

(A) Initial Version

```
main(X,Y) ->
  A = add(X,Y),
  D = dif(X,Y),
  A * D.
add(X,Y) ->
  X + Y.
dif(X,Y) ->
  X - Y.
```

(B) Program well refactored

```
main(X,Y) ->
  A = add(X,Y),
  D = dif(X,Y),
  A * D.
add(X,Y) ->
  X + Y.
dif(Y,X) ->
  X - Y.
```

(C) Program bad refactored

FIGURE 8.1: Two versions of a simple Erlang program with good and erroneous refactorings

In this example, the comparison of program versions in Figures 8.1a and 8.1b reports that the executions of both versions with respect to the selected POIs behave identically. On the other hand, if we compare versions in Figures 8.1a and 8.1c, the approach detects an error introduced in the behaviour of the program, since the arguments of the function `dif` *in the bad refactored version are swapped.*

During the rest of the chapter we make a detailed description of the different processes and transformations that make POI testing possible. We start by providing an overall view of the whole POI testing methodology (Section 8.1). Then, we show the process used to trace the execution information to compare program versions (Section 8.2). After that, we describe how the methodology is adapted to Erlang (Section 8.3), including the steps needed to enhance the process to evaluate multiple POIs at once (Section 8.4) and its applicability in programs with concurrency (Section 8.5). Finally, we present how the idea of POI testing is implemented in a tool called SecEr (Section 8.6) and show the comparison of metrics about the tool's possible configurations (Section 8.7).

8.1 A novel approach to Automatic Regression Testing: Point of Interest Testing

Our technique is divided into three sequential phases that are summarised in Figures 8.2, 8.3, and 8.4. In these figures, the big dark grey areas are used to group several processes with a common objective. Light grey boxes outside these areas represent inputs and light grey boxes inside these areas represent processes, white boxes represent intermediate results, and the initial processes of each phase are represented with a bold border box.

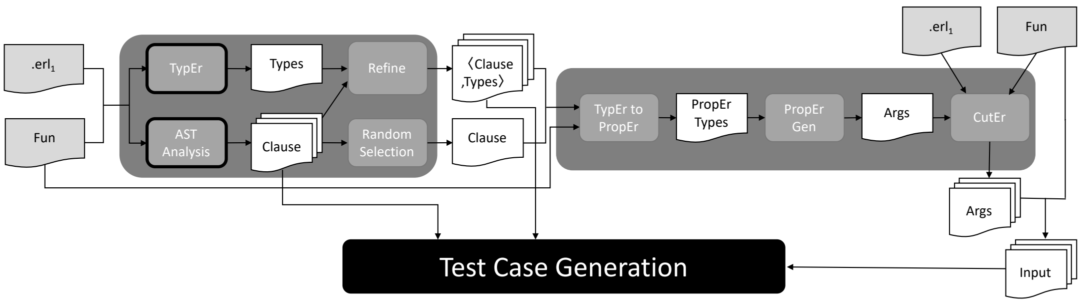


FIGURE 8.2: Type analysis phase

The first phase, depicted in Figure 8.2, is a type analysis that is in charge of preparing all inputs of the second phase (Test Case Generation). This phase starts by locating in the source code the Erlang module (`.erl`$_1$) and a function (`Fun`) specified in the user input[1] (for instance, function `exp` in the `math` module). Then, TypEr is used to obtain the type of the parameters of that function. It is important to know that, in Erlang, a function is composed of clauses and, when a function is invoked, an internal algorithm traverses all the clauses in order to select the one that will be executed. Unfortunately, TypEr does not provide the individual type of each clause, but a global type for the whole function. Therefore, we need to first analyze the AST of the module to identify all the clauses of the input function, and then we refine the types provided by TypEr to determine the specific type of each clause. All these clause types are used in the second phase. In this phase, we use PropEr to instantiate only one of them (e.g., $\langle Number,\ Integer\rangle$ can be instantiated to $\langle 4.22, 3\rangle$ or $\langle 6, 5\rangle$). However, PropEr is unable to understand TypEr types, so we have defined a translation process from TypEr types to PropEr types. Finally, CutEr is fed with an initial call (e.g., `math:exp(4.22,3)`) and it provides a set of possible arguments (e.g., $\{\langle 1.5, 6\rangle, \langle 2, 1\rangle, \langle 1.33, 4\rangle, \ldots\}$). Finally, this set is combined with the function to be called to generate the ITCs (e.g., {`math:exp(1.5,6)`, `math:exp(2,1)`, `math:exp(1.33,4)`, ...}). All this process is further detailed later.

The second phase, shown in Figure 8.3, is in charge of generating the test suite. As an initial step, we instrument the program so that its execution records (as a side-effect) the sequence of values produced at the POI defined by the user. Then, we store all ITCs provided by the previous phase into a working list. Note that it is also possible that the previous phase is unable to provide any ITC due to the limitations of CutEr. In such a case, or when there are no more ITCs left, we randomly generate a new one with PropEr and store it on the working list. Then, each ITC on the working list is processed by invoking it with the instrumented code. The execution provides the sequences of values the POI is evaluated to (i.e., the trace). This trace together with the ITC form

[1]We show here the process for only one function. In case the user defined more than one input function, the process described here would be repeated for each function.

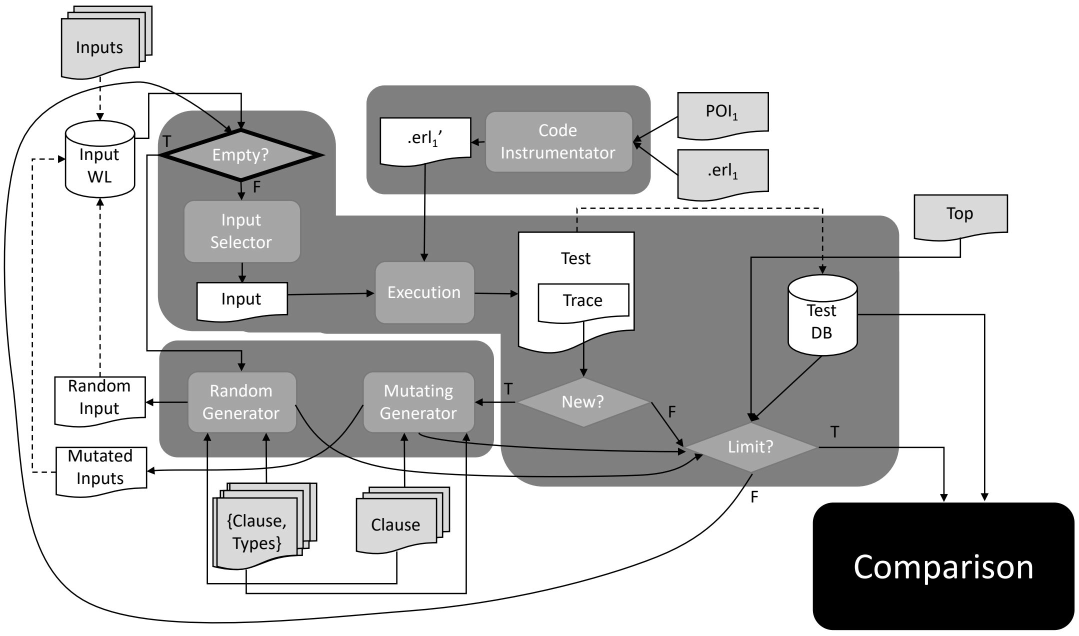


FIGURE 8.3: Test case generation phase

a new test case, which is a new output of the phase. Moreover, to increase the quality of the test cases produced, whenever a non-previously generated trace is computed, we mutate the ITC that generated that trace to obtain more ITCs. The reason is that a mutation of this ITC will probably generate more ITCs that also evaluate the POI but to different values. This process is repeated until the specified limit of test cases is reached.

Finally, the last phase (shown in Figure 8.4) checks whether the new version of the code passes the test suite. First, the source code of the new version is also instrumented to compute the traces produced at its POI. Then, all the generated test cases are executed and the traces produced are compared with the expected traces.

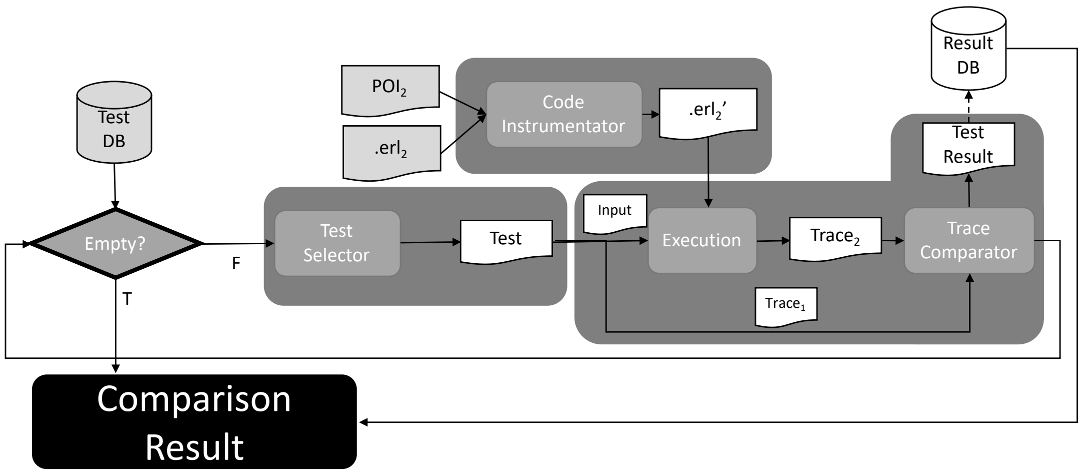


FIGURE 8.4: Comparison phase

8.2 Traced information in POI testing

POI testing uses traces of the POI to check whether its behaviour remains unchanged across program versions. We represent each element of this trace as a triplet $(POI, value, ai)$. In this triplet, POI represents the selected point of interest, $value$ represents the value computed for POI in a specific execution, and ai is a map that contains extra information about the program context for this particular execution of POI. Then, any implementation of POI testing must be able to build these triplets, storing in ai all the desired execution information (e.g. the arguments of the call when the POI is a function call), and put all the collected traces together for its usage during the comparison and the report stages.

The comparison of the traces generated during the execution is a key step to detect if two versions of a program present any unexpected behaviour. In many different situations, the information we want to compare between both executions may not be only the value produced for a POI, but different aspects of the execution like the execution time of both code versions. For this reason, POI testing must allow the usage of any customised comparison function. This feature gives users a complete freedom to configure the testing and/or debugging process in the best way according to their needs.

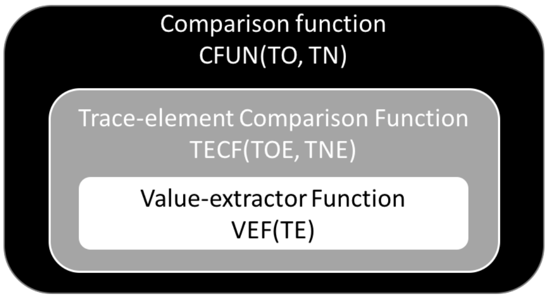


FIGURE 8.5: Comparison function structure

In order to maximise the customisation level of the comparison, users can define their own comparison functions. Each comparison function is defined by a set of connected functions that cover the different aspects of the comparison. Thus, the input part of the approach includes the definition of two functions. The connection between these functions is illustrated in Figure 8.5. The outer black box represents the general comparison function (it receives the whole traces of both executions as parameters, `TO` and `TN`), while the gray and white boxes represent the mentioned functions, that are used inside the general comparison to separately treat the elements of the whole execution trace. Their behaviour is explained below[2].

- **Value-Extractor Function (`VEF`)**: This function works at the trace element level. Its target is to extract for any trace element only the parts that the user wants to compare. For example, function:

[2]All the presented functions are written in pseudocode. In a particular implementation, all functions should be implemented in the target language.

```
VEF(POI, value, ai) ⇒ RETURN (POI, value, ai(args))
```

extracts from the additional information of a trace element only the value of the arguments (available only when the POI is placed in a method call) and ignores the rest of additional information[3].

- **Trace Element Comparison Function (TECF)**: In order to allow users to check mismatching traces found in different ways and not only a plain equality function, i.e. operator `==`, we add a comparison function for each pair of trace elements. This function iteratively receives pairs of trace elements contained in the whole traces, and it is the one in charge of comparing them. In order to compare two trace elements, it uses the `VEF` function to extract their values, perform a defined comparison, and, if a mismatching trace is found, it returns an error type notifying about it. For example, function:

  ```
  TECF(TOE, TNE) ⇒
      CASE compare(VEF(TOE),VEF(TNE)) OF
          gt → RETURN true
          eq → RETURN same
          lt → RETURN downgrade
      ENDCASE
  ```

  is an example of a `TECF`, where function `compare/2` is used to check whether a reduction in some performance indicator is obtained. Then, when it is not obtained, either a `same` or a `downgrade` labeled error type is returned. These labeled errors can trigger a customised error message defined by the user with the trace information considered relevant.

Additionally, when the comparison function detects a labelled error type, a specific report can be generated. In order to further define these error messages, we have added to the approach a way to specify how the POI tester should react to a particular unexpected behaviour. In this case, we have added an extra input parameter, a report map (RM) to link error types with error messages called. This mapping associates each error label to a corresponding error message written by the user, which can be simply a text message or contain extra information about the previously traced values. For example, expression `RM(downgrade)` may return a function similar to the one shown in Figure 8.6, where a custom message is shown[4] when an unexpected behaviour of this type is found during the execution.

[3]We use the notation *ai(key)* to refer to access some specific information previously stored in the ai map. In this case *ai(args)* represent the arguments of a POI placed in a function call.

[4]The function presented in Figure 8.6, does not make use of parameters `TEO`, `TEN` and `History` to define the message content. However, more complex functions that treat the information stored in these parameters can be defined to obtain more elaborated messages.

```
DOWNGRADE(TEO, TEN, History) ⇒
    RETURN "There has been a downgrade in the new version"
```

FIGURE 8.6: Example of a function providing a customised error message

8.2.1 Possible POI testing configurations

There are several ways of using the additional information stored in the trace elements, and all these modes are defined by the added resources previously introduced (`TECF` and `VEF`). We show three different modes which will be more useful for users:

1. **Not using additional information during the comparison.** In this mode, the traced values are the only data used when comparing the trace elements. This mode will use a value-extractor function that ignores the additional information element of the triplet

   ```
   VEF(POI, V, AI) ⇒ RETURN (POI, V)
   ```

   According on whether additional information is used for error classification or error report, we have identified three submodes.

   (a) **Additional information is only used to define error types.** In this case, additional information is only used to define new types of error labels. This mode is really useful when the additional information is not useful by itself during the comparison, but can be used to classify an unexpected behaviour detected to ease the bug location. For instance, defining a new type of labelled error `diff_value_same_args` for function calls gives the user information about where the values of two POIs placed at function calls differ, pointing out that the error is not in the call arguments, but in the implementation of the function.

   Figure 8.7 shows a `TECF` which distinguishes between those unexpected values for call POIs where the arguments are the same and those where the arguments are different[5].

```
TECF(TOE, TNE) ⇒
    IF VEF(TOE) == VEF(TNE) THEN
        RETURN true
    ELSE
        IF get_ai(TOE)(args) == get_ai(TNE)(args) THEN
            RETURN diff_value_same_args
        ELSE
            RETURN diff_value_diff_args
        ENDIF
    ENDIF
```

FIGURE 8.7: TECF which returns different error types

[5]Function `get_ai` is defined as `get_ai(POI,value,ai) ⇒ RETURN ai`.

   (b) **Additional information only used in the report stage.** If we consider that the additional information is not representative enough to categorise new types of errors, we can use its data only in the reports. This is a less intrusive way of using the additional information, but still a useful way to obtain richer feedback in the final report of each mismatching trace.

   (c) **Additional information is used to categorise and report errors.** This submode takes the advantages of both previous submodes. It also involves specific trace-element comparison functions and additions in the mismatching trace reports.

2. **Using additional information during the comparison.** This mode is the one that gives a major relevance to the additional information. By using this mode, the value and the additional information is compared as a whole. This means that, for instance, even if the compared values are the same, when any pair of elements of the additional information differs, the ITC is reported to be generating an unexpected behaviour. This mode is very convenient to uncover some errors earlier. It can also be used for performance checking, e.g. the values of the trace elements are equal but a performance indicator included in the additional information is revealing some downgrade. This mode uses a VEF function and a TECF which takes into account all or some parts of the additional information contained in each trace element.

3. **Additional information traced independently.** Finally, in this completely different mode, the additional information is considered as a separated entity and constitutes a single trace element as the ones that are generated for the POIs. This mode is very convenient in such cases where the additional information can be directly used to uncover an unexpected behaviour, avoiding in this way the comparison of several subcomputations. For instance, if we place a POI in a call, and the call parameters are compared before comparing the call result, all intermediate trace elements are not compared. This mode can be combined in such a way that other additional information is attached to these special trace elements forming a hybrid mode suitable for some specific scenarios.

8.3 POI testing adapted to Erlang

Now that we have seen an overview of how POI testing works and the components forming the traced information, we describe in more detail some relevant parts of our approach and how we have adapted POI testing to Erlang.

8.3.1 Initial ITC generation

The process starts from the type inferred by TypEr for the whole input function. This is the first important step to obtain a significant result, because ITCs are generated with the types returned by this process, so the more accurate the types are, the more accurate the ITCs will be. The standard output of TypEr is an Erlang type specification returned as a string, which would need to be parsed. For this reason, we have hacked the Erlang module that implements this functionality to obtain the types in a data structure, easier to traverse and handle. In order to improve the accuracy, we define a type for each clause of the function ensuring that the later generated ITCs will match it. For this reason, TypEr types need to be refined to TypEr types per clause.

However, the types returned by TypEr have (in our context) two drawbacks that need to be corrected since they could yield to ITCs that do not match a desired input function. These drawbacks are due to the occurrence of repeated variables and due to the type produced for lists. We explain both drawbacks with Example 8.2.

Example 8.2. *Consider a function with the header `f(A,[A,B])` as its only clause. Consider also the case where, for this function, TypEr returns the following types: `f(1 | 2, [1 | 2 | 5 | 6, ...])`* [6].

- ***Drawback 1.*** *The first drawback is caused by the fact that the value relation generated by the repeated variable `A` is lost in the function type. In particular, the actual type of variable `A` is diluted in the type of the second argument. The use of the inferred types could yield to mismatching ITCs, e.g., `f(1,[6,5])`.*

- ***Drawback 2.*** *The type of the second parameter of function `f/2` indicates that the feasible values for the second parameter are proper lists with a single constraint: each list has to be formed by numbers from the set `{1,2,5,6}`. This introduces a limit about possible values, but it does not introduce a limit to the number of elements the list must contain (two in the example). If we use the TypEr inferred types directly, we may generate ITCs that will not match the function, e.g. `f(2,[2,1,6,5])`.*

As seen in Example 8.2, the types produced by TypEr are too imprecise in our context, because they may produce test cases that are useless (e.g., non-executable). These problems are resolved along the ITC generation process. With the information we have so far, there is a problem that we can already solve, the problem arisen by *Drawback 1*, introduced by repeated variables, such as the `A` variable in the example. To solve *Drawback 1*, we traverse the parameters building a correspondence between each variable and the inferred TypEr type. Each time a repeated variable name appears, we calculate its type as the intersection of both the TypEr type and the accumulated type. For instance, in the previous example we have `A = 1 | 2` for the first occurrence, and `A = 1 | 2 | 5 | 6` for the second one, obtaining the new accumulated type `A = 1 | 2`.

[6]Erlang is a dynamically typed language and, for this reason, TypEr uses a *success typing* system instead of the usual Hindley-Milner type inference system. Therefore, TypEr's types are different from what many programmers would expect (i.e., Integer, String, etc.). Instead, a TypEr's type is a set of values such as `[ 1 | 2 | 5 | 6 ]` or an Erlang defined type (Number, Integer, etc.).

Once we have our refined TypEr types, we rely on PropEr to obtain the input for CutEr. PropEr is a property-based testing framework with a lot of useful underlying functionality. One of them is the term generators, which, given a PropEr type, are able to randomly generate terms belonging to such type. Thus, we can use the generators in our framework to generate values for a given type.

However, TypEr and PropEr use slightly different notations for their types, something reasonable given that their scopes are completely different. Unfortunately, there is not any available translator from TypEr types to PropEr types. In our technique, we need such a translator to link the inferred types to the PropEr generators. Therefore, we have built the translator by ourselves. Then, during the variable-value generation process, we deal with *Drawback 2*. In this case, we make use of the information given by the parameters of the clause in conjunction with their types. Each time a list is found during the variable-value generation, we traverse its elements and generate a type for each element on the list. Thereby, we translate an undefined sized list given by TypEr to a list with exactly the same number of elements. During this process, the scenario generated by *Drawback 1* is solved by using a dictionary of already generated variables with pairs variable-value. Each time a repeated variable is found we use the dictionary to replicate the value instead of generating a new one.

We can feed CutEr with an initial call by using a randomly selected clause and the values generated by PropEr for this clause. CutEr is a concolic testing framework that generates a list of arguments that tries to cover all the execution paths. Unfortunately, this list is only used internally by CutEr, so we have hacked CutEr to extract all these arguments. Finally, by using this slightly modified version of CutEr we are able to mix the arguments with the input function to generate the initial set of ITCs.

8.3.2 Recording the traces of the point of interest

There exist several tools available to trace Erlang executions [45, 51, 53, 194]. However, none of them allows for defining an arbitrary expression of the code. Being able to trace any possible point of interest requires either a code instrumentation, a debugger, or a way to take control of the execution of Erlang. However, using a debugger (e.g., [51]) has the drawback that it does not provide a value for the POI when it is inside an expression whose whole evaluation fails. Therefore, we decided to instrument the code in such a way that, without modifying the semantics of the code, traces are collected as a side effect when executing any (sub)expression of the code.

The instrumentation process creates and collects the traces of the POI regardless where the location of the selected expression. To create the traces in an automatic way, we instrument the expression pointed by the POI. To collect the traces, there are several options. For instance, it is possible to store the traces in a file and process it when the execution finishes, but this approach is inefficient. We follow an alternative approach based on message passing. We send messages to a server (which we call the *tracing server*) that is continuously listening for new traces until a message indicating the end of the evaluation is

```
1  foo(A) ->
2    {B,POI} = {2,A}.
```

```
{match,2,
  {tuple,2,
    [{var,2,‘B’},
     {var,2,‘POI’}]
  },
  {tuple,2,
    [{integer,2,2},
     {var,2,‘A’},]
  }
}
```

Figure 8.8: Code to be transformed and AST associated to the pattern matching in line 2

received. This approach is closer to the Erlang's philosophy. Additionally, it is more efficient since the messages are sent asynchronously resulting in an imperceptible overhead in the execution. As a result of the instrumenting process, the transformed code sends to the tracing server the value of the POI each time it is evaluated, and the tracing server stores the sequence of values.

In the following, we explain in detail how the communication with the server is placed in the code. This process is divided in steps and each step is applied to the code in Figure 8.8.

1. We first use the `erl_syntax_lib:annotate_bindings/2` function to annotate the AST of the code. This function annotates each node with two lists of variables: those variables that are being bound (`bound`) and those that were already bound (`free`) in its subtree. Additionally, we annotate each node with a unique integer that serves as identifier, so we call it *AST identifier*. This annotation is performed in a post-order traversal, resulting, consequently, in an AST where the root has the greatest number.

```
{match,2, %{nodeinfo,7,{bound,[‘B’,‘POI’]},{free,[‘A’]}}
  {tuple,2, %{nodeinfo,3,{bound,[‘B’,‘POI’]},{free,[]}}
    [{var,2,‘B’}, %{nodeinfo,1,{bound,[‘B’]},{free,[]}}
     {var,2,‘POI’}] %{nodeinfo,2,{bound,[‘POI’]},{free,[]}}
  },
  {tuple,2, %{nodeinfo,6,{bound,[]},{free,[‘A’]}}
    [{integer,2,2}, %{nodeinfo,4,{bound,[]},{free,[]}}
     {var,2,‘A’}] %{nodeinfo,5,{bound,[]},{free,[‘A’]}}
  }
}
```

Figure 8.9: Sets of environment variables and variables being bound in each expression of the AST in Figure 8.8

2. The next step is to find the POI selected by the user in the code and obtain the corresponding AST identifier. There are two ways of doing this depending on how the POI is specified: (i) If the POI is defined with

the triplet *(line, type of expression, occurrence)*, we locate it with a pre-order traversal[7] of the tree. However, (ii) when the POI is defined with the initial and final positions, we replace, in the source code, the whole expression with a fresh term. Then, we build the AST of this new code and we search for the fresh term in this AST recording the path followed. This path is replicated in the original AST to obtain the AST identifier of the POI. Thus, the result of this step is a relation between a POI and an AST identifier. According to the annotated AST given in Figure 8.9 the result of this step would be $\langle$`POI,2`$\rangle$.

3. Then, we need to extract the path from the AST root to the node with the AST identifier with a new search process. During this search process, we store the path followed in the AST with tuples of the form `(Node, ChildIndex)`, where `Node` is the AST node and `ChildIndex` is the index of the node in its parent's children array. Obtaining this path is essential for the next steps since it allows us to recursively update the tree in an easy and efficient way. When the POI is found, the traversal finishes. Thus, the output of this step is a path that yields directly from the root node to the POI. The path computed for POI in Figure 8.8 would be the following list of tuples:

   [`{match,1},{tuple,1},{POI,2}`]

4. Most of the times, the POI can be easily instrumented by adding a send command to communicate its value to the tracing server. However, when the POI is in the pattern of an expression, this expression needs a special treatment in the instrumentation. This is shown with Example 8.3.

   Example 8.3. *Consider a POI inside a pattern* `{1,POI,3}` *of a match expression. If the execution tries to match it with* `{2,2,3}` *nothing is send to the tracing server because the POI is never evaluated. Contrarily, if it tries to match it with* `{1,2,4}` *we send the value* `2` *to the tracing server. Note that the matching fails in both cases, but due to the evaluation order, the POI is actually evaluated (and it succeeds) in the second case. There is an interesting third case, that happens when the* `POI` *has a value, e.g.,* `3`*, and the matching with* `{1,4,4}` *is tried. In this case, although the matching at the* `POI` *fails, we send the value* `4` *to the tracing server. We could also send its actual value, i.e.,* `3`*. This is just a design decision, but we think that including the value that produced the mismatch could be more useful to find the source of a discrepancy.*

   We call *target expression* to those expressions that need a special treatment in the instrumentation as the previously described one. In Erlang, these target expressions are: pattern-matchings, list comprehensions, and expressions with clauses (i.e., `case`, `if`, `functions`, ...). The goal of

[7]We use this order because it is the one that allows us to find the nodes in the same order as they are in the source code.

this step is to divide the AST path into two sub-paths `(PathBefore, PathAfter)`. `PathBefore` yields from the root to the deepest target expression (included), and `PathAfter` yields from the first children of the target expression to the POI node. The division of the path is shown hereunder, where `PathBefore` is shown in blue and `PathAfter` in red.

```
[{match,1},{tuple,1},{POI,2}]
```

5. Finally, the last step is the one in charge of performing the actual instrumentation. The `PathBefore` is used to traverse the tree until the deepest target expression that contains the POI is reached. At this point, five rules (described below) are used to transform the code by using `PathAfter`. Finally, `PathBefore` is traversed backwards to update the AST of the targeted function. The five rules are depicted in Figure 8.10. The main goal of these instrumentation rules is to introduce a send expression into the code that sends the value of the POI to the previously launched process `tracer`. The difficulty here is to prevent some Erlang features from affecting this sending like the scenario described before. The first four rules are mutually exclusive, and when none of them can be applied, the rule `(EXPR)` is applied. Rule `(LEFT_PM)` is fired when the POI is in the pattern of a pattern-matching expression. Rule `(PAT_GEN_LC)` is used to transform a list comprehension when the POI is in the pattern of a generator. Finally, rules `(CLAUSE_PAT)`[8] and `(CLAUSE_GUARD)` transform an expression with clauses when the POI is in the pattern or in the guard of one of its clauses, respectively.

   In the rules, we use the underline symbol (_) to represent a value that is not used and $\texttt{poi}_{id}$ represents the POI AST identifier (computed during step 2), which is used to link the traced value and information to a specific POI. There are several functions used in the rules that need to be introduced. Function $hd(l)$, $tl(l)$, $length(l)$ and $last(l)$ returns the head, the tail, the length, and the last element of the list l, respectively. Function $pos(e)$ returns the child index of an expression e, i.e., its index in the list of children of its parent. Function $is\_bound(e)$ returns `true` if e is bounded according to the AST binding annotations (see step 1). Function $clauses(e)$ and $change\_clauses(e, clauses)$ obtains and modifies the clauses of e, respectively. Function $fv()$ builds a free variable. Function `get_ai()`, defined by the user, implements the necessary instrumentation to extract from the execution the selected additional information associated to the POI.

   Finally, there is a key function named pfv, introduced in Figure 8.11, that transforms a pattern so that the constraints after the POI do not inhibit the sending call. This is done by replacing all the terms on the right of the POI with free variables that are built using fv function. Unbound variables on the left and also in the POI are replaced by fresh variables to avoid the

[8]Function clauses need an additional transformation that consists in storing all the parameters inside a tuple so that they could be used in case expressions.

$$\begin{array}{rl}
\texttt{(LEFT\_PM)} & \texttt{p = e} \Rightarrow \texttt{p = begin np = e, tracer!\{add, poi}_{id}\texttt{, npoi, get\_ai()\},} \\
& \qquad\qquad\quad \texttt{np end} \\
\text{if} & \quad (\texttt{p = e}, \_\,) = \mathit{last}(\mathit{PathBefore}) \\
& \wedge\, (\,\_\,, \mathit{pos}(\texttt{p})) = \mathit{hd}(\mathit{PathAfter}) \\
\text{where} & (\,\_\,, \texttt{npoi, np}) \;=\; \mathit{pfv}(\texttt{p}, \; \mathit{PathAfter})
\end{array}$$

$$\begin{array}{rl}
\texttt{(PAT\_GEN\_LC)} & \texttt{[e || gg]} \Rightarrow \texttt{[e || ngg]} \\
\text{if} & \quad (\texttt{[e || gg]}, \_\,) = \mathit{last}(\mathit{PathBefore}) \\
& \wedge\, (\,\_\,, \mathit{pos}(\texttt{p\_gen})) = \mathit{hd}(\mathit{tl}(\mathit{PathAfter})) \\
& \wedge\, \exists\, i.\; 1 \;\leq\; i \;\leq\; \mathit{length}(\texttt{gg}) \text{ s.t. } \texttt{gg}_i = \texttt{p\_gen <- e\_gen} \\
\text{where} & \quad (\,\_\,, \texttt{npoi}, \texttt{np\_gen}) = \mathit{pfv}(\texttt{p\_gen}, \; \mathit{tl}(\mathit{PathAfter})) \\
& \wedge\, \texttt{ngg}_i = \texttt{p\_gen <- begin tracer!\{add, poi}_{id}\texttt{, npoi, get\_ai()\},} \\
& \qquad\qquad\qquad\qquad\qquad\quad \texttt{[np\_gen] end} \\
& \wedge\, \texttt{ngg} = \texttt{gg}_1 \ldots \texttt{gg}_{i-1}, \texttt{np\_gen <- e\_gen}, \; \texttt{ngg}_i, \; \texttt{gg}_{i+1} \ldots \texttt{gg}_{\mathit{length}(\texttt{gg})}
\end{array}$$

$$\begin{array}{rl}
\texttt{(CLAUSE\_PAT)} & \texttt{e} \Rightarrow \mathit{change\_clauses}(\texttt{e}, \; \texttt{ncls}) \\
\text{if} & \quad (\texttt{e}, \; \_\,) = \mathit{last}(\mathit{PathBefore}) \\
& \wedge\, (\,\_\,, \mathit{pos}(\texttt{p\_c})) = \mathit{hd}(\mathit{tl}(\mathit{PathAfter})) \\
& \wedge\, \exists\, i.\; 1 \;\leq\; i \;\leq\; \mathit{length}(\texttt{cls}) \text{ s.t. } \texttt{cls}_i = \texttt{p\_c when g\_c -> b\_c} \\
\text{where} & \quad \texttt{cls} = \mathit{clauses}(\texttt{e}) \\
& \wedge\, (\,\_\,, \; \texttt{npoi, np\_c}) = \mathit{pfv}(\texttt{p\_c}, \; \mathit{tl}(\mathit{PathAfter})) \\
& \wedge\, \texttt{nb\_c = begin tracer!\{add, poi}_{id}\texttt{, npoi, get\_ai()\},} \\
& \qquad\qquad\qquad\quad \texttt{case np\_c of cls end end} \\
& \wedge\, \texttt{ncls}_i = \texttt{np\_c when true -> nb\_c} \\
& \wedge\, \texttt{ncls} = \texttt{cls}_1, \ldots, \texttt{cls}_{i-1}, \texttt{ncls}_i, \texttt{cls}_{i+1}, \ldots, \texttt{cls}_{\mathit{length}(\texttt{cls})}
\end{array}$$

$$\begin{array}{rl}
\texttt{(CLAUSE\_GUARD)} & \texttt{e} \Rightarrow \mathit{change\_clauses}(\texttt{e}, \; \texttt{ncls}) \\
\text{if} & \quad (\texttt{e}, \; \_\,) = \mathit{last}(\mathit{PathBefore}) \\
& \wedge\, (\,\_\,, \mathit{pos}(\texttt{g\_c})) = \mathit{hd}(\mathit{tl}(\mathit{PathAfter})) \\
& \wedge\, \exists\, i.\; 1 \;\leq\; i \;\leq\; \mathit{length}(\texttt{cls}) \text{ s.t. } \texttt{cls}_i = \texttt{p\_c when g\_c -> b\_c} \\
\text{where} & \quad \texttt{cls} = \mathit{clauses}(\texttt{e}) \\
& \wedge\, (\texttt{poi}, \; \_\,) = \mathit{last}(\mathit{PathAfter}) \\
& \wedge\, \texttt{nb\_c = begin tracer!\{add, poi}_{id}\texttt{, poi, get\_ai()\},} \\
& \qquad\qquad\qquad\quad \texttt{case np\_c of cls end end} \\
& \wedge\, \texttt{ncl} = \texttt{p\_c when true -> nb\_c} \\
& \wedge\, \texttt{ncls} = \texttt{cls}_1, \ldots, \texttt{cls}_{i-1}, \texttt{ncl}, \texttt{cls}_{i+1}, \ldots, \texttt{cls}_{\mathit{length}(\texttt{cls})}
\end{array}$$

$$\begin{array}{rl}
\texttt{(EXPR)} & \texttt{e} \Rightarrow \texttt{begin fv = e, tracer!\{add, poi}_{id}\texttt{, fv, get\_ai()\}, fv end} \\
\text{otherwise} & \\
\text{where} & (\texttt{e}, \; \_\,) = \mathit{last}(\mathit{PathAfter}) \;\wedge\; \texttt{fv} = \mathit{fv}()
\end{array}$$

FIGURE 8.10: Instrumentation rules for tracing

shadowing of the original variables. In the *pfv* function, $children(e)$ and $change\_children(e, children)$ are used to obtain and modify the children of expression e, respectively. In this function, lists are represented with the head-tail notation $(h : t)$.

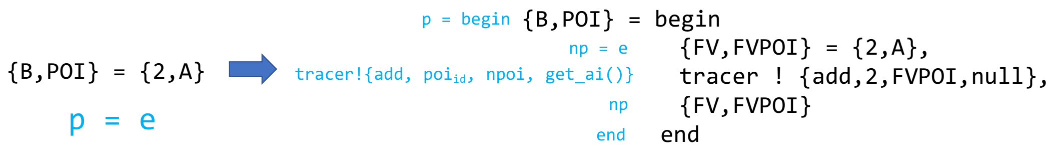
Figure 8.12 shows the application of the rule `LEFT_PM` to the code in Figure 8.8. Each fragment of the rule is shown in blue and represented next to

$$
\begin{array}{l}
pfv(p,\ path) = \\
\quad \begin{cases}
(poi,\ poi',\ p'') \quad \text{if } path = [(poi, pos)] \\
\qquad\qquad \text{where } poi' = fv() \ \wedge\ p' = fv\_from(pos,\ p) \\
\qquad\qquad\quad \wedge\ p'' = p'_1 \ldots p'_{pos-1}, poi', p'_{pos+1} \ldots p'_{length(p)} \\
(poi,\ poi',\ p''') \quad \text{otherwise} \\
\qquad\qquad \text{where } (\_, pos) = hd(path) \ \wedge\ p' = fv\_from(pos,\ p) \\
\qquad\qquad\quad \wedge\ (poi, poi', p'') = pfv(p'_{pos}, tl(path)) \\
\qquad\qquad\quad \wedge\ p''' = p'_1 \ldots p'_{pos-1}, p'', p'_{pos+1} \ldots p'_{length(p)}
\end{cases} \\
fv\_from(pos,\ p) = \\
\qquad p'_1 \ldots p'_{pos}, fv()_{pos+1} \ldots fv()_{length(p)} \text{ where } (p'_1 \ldots p'_{pos}, \_) = cv(p_1 \ldots p_{pos},\ [])
\end{array}
$$

$$
\begin{array}{l}
cv(list, map) = \\
\quad \begin{cases}
([], map) & \text{if } list = [] \\
((fv : p'_t), map') & \text{if } list = (p_h : p_t) \wedge is\_var(p_h) \wedge \neg\ is\_bound(p_h) \\
\qquad\qquad \text{where } fv = fv() \ \wedge\ (p'_t,\ map') = cv(p_t, map \cup \{p_h \mapsto fv\}) & \\
((fv_{map} : p'_t), map') & \text{if } list = (p_h : p_t) \wedge is\_var(p_h) \wedge p_h \mapsto fv_{map} \in map \\
\qquad\qquad \text{where } (p'_t, map') = cv(p_t, map) & \\
((p'_h : p'_t), map'') & \text{otherwise} \\
\qquad\qquad \text{where } (p_h : p_t) = list \ \wedge\ (children'_{p_h}, map') = cv(children(p_h), map) & \\
\qquad\qquad\quad \wedge\ p'_h = change\_children(p_h, children'_{p_h}) & \\
\qquad\qquad\quad \wedge\ (p'_t, map'') = cv(p_t, map') &
\end{cases}
\end{array}
$$

FIGURE 8.11: Function pfv

the corresponding part of the code. The application of the rule transforms one pattern matching expression into another one which right-hand side is a block. This transformation allows the tracing of the POI without modifying the semantics of the program. This is done by temporarily storing the values of the original right-hand side of the match into free variables, sending this values to the tracing server (`tracer`), and returning the free variable's pattern of the inner matching as the result of the block. It is worth to mention that to construct the sending message we extract from the annotated AST (Figure 8.9) the AST identifier of the POI, including it into the message.



```
{B,POI} = {2,A}

{B,POI} = begin
              {FV,FVPOI} = {2,A},
              tracer ! {add,2,FVPOI,null},
              {FV,FVPOI}
          end
```

FIGURE 8.12: Application of rule `LEFT_PM` to the POI in line 2 of Figure 8.8

8.3.3 Extraction of additional trace information

The possibility of including in the execution trace some extra information about the execution environment, gives versatility to the POI testing approach. Although in many cases we may only be interested in the value generated for a POI without any context information, there are other scenarios where different informations become really useful. Additional information can include performance checks between versions by including the recordings of execution time or memory usage, but there are also some program constructs where the information of POI-related (sub)expressions can be used to ease the location of the error when a mismatching trace is found. We show two examples of how to include in the trace different environment information that may be used when selecting a POI: (i) the value of the parameters when the POI is placed in a function call, and (ii) the current call stack each time the POI is evaluated.

Enhanced Method calls

When we place a POI in a call, we are saying that we are interested in comparing the result of this call, so the standard behaviour of a POI tester is to trace only these values. Nevertheless, when we detect a mismatching trace it is interesting to have an enhanced trace where not only the result of the call, but also its arguments, are traced. Therefore, we need to add to the additional information mapping a new element whose key is `ca` and whose value is a list that contains the call arguments.

In order to obtain the additional information call traces, we have to define a way for sending, receiving and merging the call result and its call arguments. The main idea is to send the argument traces before actually performing the call and the call's result just after that. Thus, we should define how this additional information is added to the POI trace. Figure 8.13a shows how this instrumentation is done for Erlang.

$$
\begin{array}{ll}
\mathtt{e}(\overline{\mathtt{e_i}}) \Rightarrow \mathtt{begin} & \\
 & \mathtt{fv_{ref}} = \mathtt{make\_ref}(), \\
 & [\mathtt{fv_v}|\overline{\mathtt{fv_{v_i}}}] \ = \ [\mathtt{e}|\overline{\mathtt{e_i}}], \\
 & \mathtt{tracer!}\{\mathtt{add\_ci}, \mathtt{POI}, \{\mathtt{fv_{ref}}, \mathtt{fv_v}\}\}, \\
 & \overline{\mathtt{tracer!}\{\mathtt{add\_ci}, \mathtt{POI}, \{\mathtt{fv_{ref}}, \mathtt{fv_i}\}\},} \\
 & \mathtt{fv} = \mathtt{fv_v}(\overline{\mathtt{fv_{v_i}}}), \\
 & \mathtt{tracer!}\{\mathtt{add}, \mathtt{POI}, \{\mathtt{fv_{ref}}, \mathtt{fv}\}\}, \\
 & \mathtt{fv} \\
 \mathtt{end} &
\end{array}
$$

(A) Instrumentation rule for call tracing

```
tracer({Stack, Trace}) ->
  receive
    {add_ci, POI, AI} ->
      tracer({[AI | Stack], Trace});
    {add, POI, {Ref, V}} ->
      {CalleeArgs, NStack} =
        remove_same_ref(Ref, Stack),
      tracer({NStack,
        [{POI, V, store(ca, CalleeArgs)}
         | Trace]});
    {add, POI, V} ->
      tracer({Stack, [{POI, V} | Trace]})
  end.
```

(B) Simplified tracing server

FIGURE 8.13: Elements of our proposed call tracer enhancement in Erlang

When the code instrumentation process finds a call, i.e. $\mathtt{e}(\overline{\mathtt{e_i}})$, the expression is then replaced by the block expression (`begin-end`) on the right-hand side. This instrumentation (i) creates a set of auxiliary free variables[9] to store all the evaluated context of the call (callee and arguments), (ii) sends to the tracer of these defined variables, (iii) performs the actual call using the value of the callee and the arguments, (iv) sends the result of the call to the trace server,

[9]All free variables used in the rule are represented as $\mathtt{fv}_*$. Each one of these free variables is unique and different to all the original variables of the module.

and (v) returns the result of the call to make the block return the expected result. This separate process needs to be done this way to avoid the multiple execution of call arguments, which may produce side-effects in some cases. It is worth mentioning the first expression of the block, which creates a unique reference (with function `make_ref/0` in Erlang) that serves to identify all the traces belonging to the same concrete execution of a call. This reference is also stored in a free variable, ($\mathtt{fv}_{\mathtt{ref}}$) and is used to link call arguments to call result at the tracing server.

All the information sent while running the instrumented code is received and merged by the tracer. In Erlang, the tracer is a server which is continuously receiving trace elements until the end of the execution or until a timeout is raised. Figure 8.13b shows a simplification of the Erlang function `tracer/1` which is in charge of this tracing process. The server state is a tuple containing: 1) a stack, where the callee and arguments are stored in the order they are received, and 2) the trace generated so far. Its body is a receive expression with three clauses: the first one is for the information sent by function calls' callees and arguments, the second one is for the result of the function call, and the third one is for the rest of trace elements, i.e. those that do not come from a function call. When a callee or an argument value is received, it is simply stacked. When the call result is received, all its arguments, which are at the top of the stack, are unstacked (function `remove_same_ref/2`), and stored in the additional information of the call trace element. Finally, the rest of trace elements are simply added to the current trace with an empty additional information structure.

Stack-recording POIs

Stack traces are common elements in error reports due to the valuable information they provide to help find the source of an error. In our context, an error is not a common bug but an unexpected behaviour. However, stack traces can be also really helpful in this context. Suppose we are performing POI testing on two programs using a single POI inside a common function that is called from different parts of the program. When POI testing is run, we can get some reports informing of some mismatching traces, for instance, that the POI values are different in a particular execution point. Then, users should start placing POIs in previous stages of the evaluation in order to find the source of the discrepancy. However, it is not clear how to proceed in this debugging process, as there is not enough information about the discrepancy's source. This discrepancy could happen for several reasons. For example, if we are using as POI a case expression, we need to explore alternative ways to provide users with some evaluation context when any mismatching trace is found. By including the stack trace to the unexpected behaviour reports, users can check whether both versions have perform the same calls or not, i.e. they have followed parallel paths. Even when the top of the stack trace is a call with the same arguments, the produced value can differ simply because some of the elements of the rest of the stack trace differ. The discrepancy in the followed paths can be, e.g. due to impure features of the language, like, for instance, the process dictionary in Erlang. Thus, it is not a simple task to identify these discrepancies. With the

$$\begin{array}{lll} \texttt{e}(\overline{\texttt{e}_\texttt{i}}) & \Rightarrow & \texttt{begin} \\ & & \quad [\texttt{v}|\overline{\texttt{v}_\texttt{i}}] \;=\; [\texttt{e}|\overline{\texttt{e}_\texttt{i}}], \\ & & \quad \boxed{[\texttt{v}(\overline{\texttt{v}_\texttt{i}})], [\texttt{v}|\overline{\texttt{v}_\texttt{i}}]} \\ & & \texttt{end} \end{array}$$

(A) Calls

$$\texttt{f}(\overline{\texttt{p}_\texttt{i}}) \to \overline{\texttt{e}_\texttt{i}}. \quad \Rightarrow \quad \texttt{f}(\overline{\texttt{p}_\texttt{i}}) \to \boxed{\overline{\texttt{e}_\texttt{i}}, [\texttt{f}|\overline{\texttt{p}_\texttt{i}}]}$$

(B) Function definitions

$$\begin{array}{lll} \boxed{\overline{\texttt{e}_\texttt{i}}, \texttt{call}} & \Rightarrow & \texttt{begin} \\ & & \quad \texttt{fv}_\texttt{ref} = \texttt{make\_ref}(), \\ & & \quad \texttt{tracer!}\{\texttt{begin}, \texttt{fv}_\texttt{ref}, \texttt{call}\}, \\ & & \quad \texttt{try} \\ & & \quad\quad \overline{\texttt{e}_{\texttt{i}-\texttt{1}}}, \\ & & \quad\quad \texttt{fv} = \texttt{e}_\texttt{i}, \\ & & \quad\quad \texttt{tracer!}\{\texttt{end}, \texttt{fv}_\texttt{ref}\}, \\ & & \quad\quad \texttt{fv} \\ & & \quad \texttt{catch} \\ & & \quad\quad \texttt{E}:\texttt{R} \to \texttt{tracer!}\{\texttt{end}, \texttt{fv}_\texttt{ref}\}, \\ & & \quad\quad\qquad \texttt{error}(\texttt{E}, \texttt{R}) \\ & & \quad \texttt{end} \\ & & \texttt{end} \end{array}$$

(C) Stack trace template

FIGURE 8.14: Transformation rules to obtain the stack trace

addition of the stack trace information, users can go directly to the function that start creating the bifurcation on the paths and start a debugging process there. By using special comparison functions, this enhancement can be useful even when some renaming or refactoring process has been performed.

In a similar way as we did with function calls, we need to add stack traces to the each POI evaluation. This again involves modifying sending and reception of trace elements. However, in this case, the modifications needed are much simpler. First, the rules used by the instrumentation process need to augment each message send to the tracer with the stack trace. In Erlang, this is done by using the `catch _:_:ST -> ...` catch clause pattern when an exception has been thrown[10]. On the other hand, the reception should be adapted accordingly to process these new trace elements. This involves, modifying the receive's clauses. In concrete, we can use a single clause like the one in lines 11-12 of Figure 8.13b:

```
{add, POI, V, ST} ->
        tracer([{POI, V, store(st, ST)} | Trace]})
```

However, the most challenging part here is not modifying the POI testing approach, but getting useful stack traces. Some programming language, like Erlang, perform *last call optimization* (LCO) which solve important performance issues. However, LCO comes with an important drawback: the stack traces that are reported in errors can be incomplete. This can be very confusing and annoying for users when they use reported stack traces that could be used during the debugging of a buggy code. Nevertheless, there are ways to inhibit LCO. Of course, this should be done very carefully, as its impact in the performance can be disastrous. For instance, a program transformation that changes the code in a way that the last expression of function bodies is never a call. In this way, we can get full stack traces by using the standard methods provided by

[10]This pattern is only allowed in the handler of a try-catch expression. We do not further describe the instrumentation process used to extract this stack trace for the sake of simplicity.

```
1  stack_tracer(Stack) ->
2      receive
3          {begin, Ref, Call} ->
4              stack_tracer([{Ref, Call} | Stack]);
5          {end, Ref} ->
6              case Stack of
7                  [{Ref, _} | T] ->
8                      stack_tracer(T);
9                  [_ | T] ->
10                     NStack =
11                         unstack_till(T, Ref)
12                     stack_tracer(NStack)
13             end
14     end.
```

FIGURE 8.15: Stack tracer

the language. An alternative is to forget these standard methods and *manually* build the stack trace during the POI tracing instrumentation. With this approach, the stack trace is dynamically built and each time a POI trace is sent, a snapshot of the current stack (which is part of the server's state) is stored in the additional information mapping. Finally, there is another alternative which does not include the stack trace during the testing process. Instead, before UBs are reported, their correspondent ITCs are rerun sending the stack trace for only a specific execution of a POI, e.g. the fourth time it is executed.

Both alternatives, LCO inhibition and manual stack trace building, can be useful tools to get the full stack trace, especially in those cases where the expected size of the stack traces are not huge. In the case of Erlang, by inhibiting LCO we get the standard stack trace provided by Erlang, which does not include all the call arguments, but just the callee. We can improve this in the manual version. Figure 8.14 shows two transformation rules that can be used to manually obtain stack traces. This transformation should be done after the POI instrumentation in order to be correct. Both rules use a template (represented by $\boxed{\overline{\mathtt{e_i}}, \mathtt{call}}$) to create stack traces. The idea behind this template is to send a `begin` trace just before starting the call and an `end` trace just after. This can be done in the calls (Figure 8.14a) or/and in the function definitions (Figure 8.14b[11]). By transforming only the calls we can stack all the calls performed in user-defined code, even calls to external libraries. However, calls to user function performed from a non-user-defined code, e.g. from a `lists:map/2`, would not be stacked. By transforming only function definitions this benefit and drawback are reversed. Using both at the same time is probably the best choice. However, this configuration produces duplicated `begin-end` traces when a user-defined function is called from user-defined code. This is solved at the tracer side. Figure 8.15 shows a simplification of a stack tracer whose state is the current stack, i.e. `Stack`. The most interesting part is how the `end` traces are processed. Clause of lines 7-8 represents the case where the top of the stack

[11]The rule shows how a clause of a function definition is transformed. A whole function definition is transformed by applying this rule for each of its clauses.

coincides with the `end` trace, i.e. a successfully finished call. On the other hand, clause of lines 9-12, represents calls where some error has been raised during their evaluation. Function `unstack_till/2` unstack elements of the stack until the expected reference, i.e. `Ref`, is found. Errors raised during a call evaluation are the reason why the call is put inside a `try-catch` expression in the transformation rule shown in Figure 8.14a. The handler of this `try-catch` expression sends an `end` trace to inform the tracer that some error has occurred.

8.3.4 Test case generation using ITC mutation

The test case generation phase uses CutEr because its concolic analyses try to generate 100% branch coverage test cases. However, sometimes these analyses require too much time and we have to abort its execution. This means that after executing CutEr, we might have no test cases. Moreover, CutEr is not exhaustive enough when it evaluates a single branch. Therefore, in our context, it is insufficient to detect behaviour differences in a concrete branch, e.g. a wrong operator.

$$
tgen(top, pending, map) = \begin{cases}
map & \text{if } size(map) \geq top \\
tgen(top, pending', map') & \text{if } size(map) < top \\
& \wedge\ \exists\ input \in pending \\
& \quad \text{s.t. } trace(input) \mapsto \_ \notin map \\
\quad \text{where } pending' = (pending \cup mut(input)) \backslash \{input\} & \\
\quad\quad \wedge\ map' = map \cup \{trace(input) \mapsto \{input\}\} & \\
tgen(top, \{proper\_gen()\}, map') & \text{if } size(map) < top \\
& \wedge\ \nexists\ input \in pending \\
& \quad \text{s.t. } trace(input) \mapsto \_ \notin map \\
\quad \text{where } map' = map & \\
\quad\quad \cup\ \{trace(input_p) \mapsto (\{input_p\} \cup inputs_{tp}) & \\
\quad\quad\quad \mid input_p \in pending \wedge trace(input_p) \mapsto inputs_{tp} \in map\} &
\end{cases}
$$

FIGURE 8.16: Test generation function

Therefore, we should produce more test cases to increase the reliability of the test suite. One option is to use the PropEr generator to randomly synthesize new test cases, but this would produce many useless test cases (e.g., test cases that do not execute the POI). Hence, in order to avoid a completely random test case generation, we use a test mutation technique. The function that generates the test cases is depicted in Figure 8.16. The result of the function is a map from the different obtained traces to the set of ITCs that produce them. The first call to this function is $tgen(top, cuter\_tests, \emptyset)$, where top is a user-defined limit of the desired number of test cases[12] and $cuter\_tests$ are the test cases that CutEr generates. Function $tgen$ uses the auxiliary functions $proper\_gen$, $trace$, and mut. The function $proper\_gen()$ simply calls PropEr to generate a new test case, while function $trace(input)$ obtains the corresponding trace when the ITC $input$ is executed. The size of a map, $size(map)$, is the total amount of elements stored in all lists that belong to the map. Finally, function $mut(input)$ obtains

[12]In SecEr, a timeout is also used as a way to stop the test case generation.

a set of mutations for the ITC *input*, where, for each argument in *input*, a new test case is generated by replacing the argument with a randomly generated value (using PropEr) and leaving the rest of arguments unchanged.

8.4 POI testing with multiple POIs

The previous sections introduced a methodology to automatically obtain traces from a given POI. An extension of this methodology to multiple POIs enables several new features like a fine-grained testing, or checking multiple functionalities at once. However, it introduces new challenges to be overcome.

In order to extend the approach for multiple POIs, we need to perform some modifications in some of the steps of the single-POI approach. The flow is exactly the same as the one depicted in Section 8.1, but we need to modify some of its internals. There is no need for modifications along the process of the ITC generation described in Section 8.3.1, since this process depends on the input functions that, in our approach, are shared by all the POIs. On the other hand, we need to adapt the tracing and mutation processes to work with the multiple POI approach.

8.4.1 Code transformation with multiple POIs

The tracing method introduced in Section 8.3.2 needs to be slightly redefined here. This section defined 5 steps that started from a source code and a POI and ended in an instrumented version of the source code that is able to communicate traces. Therefore, the only change needed is that, instead of having only one POI, we have more than one. In order to deal with this change, we follow the same 5 steps but change the way in which they are applied. In the single-POI approach, they are applied sequentially, but here we need to iterate some of them. In concrete, steps 1 and 2 are done only once in the whole process while the rest of steps are done once per POI. The result of step 2 is now a set of *POI-AST identifier* relations instead of a single one. Then, we iterate the obtained AST identifiers applying steps 3, 4, and 5 sequentially. Note that, although the result of step 5 is a new AST, we are still able to find the AST identifiers of the subsequent POIs since the transformations do not destroy any node of the original AST, instead they only move them inside a new expression. This justifies the double search design performed in steps 2 and 3. If we tried to search for the POI in a modified AST we could be unable to find it. In contrast, AST identifiers ensure that it can always be found.

In the multiple-POIs approach, there is also a justification of why the identifiers are numbers and why the identification process is done with a post-order traversal. First of all, there is one question that should be discussed: is the order in which the POIs are processed important? The answer is yes, because the user could define a POI that includes another POI inside, e.g. POI_1 is the whole tuple `{X, Y}` and POI_2 is `X`. This scenario would be problematic when the POI-inside-POI case occurs inside a pattern due to the way we instrument the code. If we instrumented first POI_1, its trace would be sent before the one of POI_2. Note that this is not correct since POI_2 is evaluated first, therefore it should be traced first. This justifies the use of a post-order traversal, where the identifier of a node is maximal in its subtree. Thus, as the AST identifiers are numbers, and their order is convenient in our context, we can order the AST identifiers obtained from the POIs before starting the transformation loop.

The test case generation phase is also affected by the inclusion of multiple POIs. In the original definition the traces were a sequence of values, therefore it was easy to check whether a trace had appeared in a previously executed test. However, with multiple POIs the trace is not such a simple sequence, since the same POI in one program version can be associated to more than one POIs on the other version of the code. Therefore, we need a more sophisticated way to determine the equality of the traces.

8.4.2 Differences in trace equality

Several alternatives can be considered when comparing traces that contain values from multiple POIs. In concrete, we propose the three different comparison functions provided to deal with this situation in Erlang.

A trace of a POI is defined as the sequence of values (together with some context information) that the POI is evaluated to during an execution. It has been represented with $trace(input)$, which obtains the corresponding trace when the ITC is executed. In the case of multiple-POI approach, the whole trace will contain a sequence of tuples of the form $(POI, value, AI)$ traced for all the POIs defined by the user, preserving their execution order. Since the POIs can be executed in any order, we need a way to compare this unsorted list of traces. Therefore, we additionally need to define a relation between POIs. This relation, that we represent with R_{POIs}, is automatically built from the input provided by the user. It is a set that contains tuples of the form (POI_{old}, POI_{new}). Once we have defined these components, we can compare traces in different ways according to our comparison goal:

1. The first of the three mentioned comparison approaches is the one that uses a simple equality function to compare two traces obtained from different versions of a program which equality function is defined as follows:

$$
equal(trace_{old}, trace_{new}, R_{POIs}) = \\
\begin{cases}
true & \text{if } trace_{old} = [] \land trace_{new} = [] \\
equal(trace'_{old}, trace'_{new}, R_{POIs}) & \\
 & \text{if } trace_{old} = (elem_{old} : trace'_{old}) \\
 & \land\ trace_{new} = (elem_{new} : trace'_{new}) \\
 & \land\ TECF(elem_{old} = elem_{new}) \\
 & \land\ elem_{old} = (POI_{old}, \_, \_) \\
 & \land\ elem_{new} = (POI_{new}, \_, \_) \\
 & \land\ (POI_{old}, POI_{new}) \in R_{POIs} \\
false & \text{otherwise}
\end{cases}
$$

Note that the standard equality function is perfectly valid for comparing traces during the test case generation phase, because all of them come from the same source code. However, it is no longer valid for comparing program versions since POIs can differ in the original program and the modified one. This equality function is useful when the user is interested in comparing the traces interleaved (i.e., when their interleaved execution is relevant). However, in some scenarios the user can be interested in relaxing the interleaving constraint and compare the traces independently.

2. The application of a relaxed comparison yields to the second comparison approach, which consists in building a mapping from POIs to sequences of POI traces in the following way:

   $$trace(input, POI) = [\,VEF((POI, v, ai)) \mid (POI, v, ai) \in trace(input)]$$

   The order is assumed to be preserved in the produced sequences. Using this sequences we can define an alternative equality function as follows:

   $$equal(trace_{old}, trace_{new}, R_{POIs}) =$$

   $$\bigwedge_{(POI_{old}, POI_{new}) \in R_{POIs}} CFUN(trace(input, POI_{old}), trace(input, POI_{new}))$$

   This kind of comparison is suitable when the cardinality of the POI relation is 1-to-1 and there is no connexion between the selected POIs, i.e., the order they are executed is completely irrelevant.

3. There is a third equality relation that could be useful in certain cases. Suppose that we detect some duplicated code, so we build a new version of the code where all the repeated code has been refactored to a single code. If we want to test if the behaviour is kept, we need to define a relation where multiple POIs in the old version are associated with a single POI in the new version. This is represented in our approach adding to the R_{POIs} several tuples of the form (POI_{old_1}, POI_{new}), (POI_{old_2}, POI_{new}), etc. A similar scenario can happen when a functionality of the original code is split in several parts in the new code. In both cases, a special treatment is needed for this type of relations. In order to do this, we define a generalisation of the previous *equal* function where this kind of relations is taken into account. The first step is to extract all the POIs in R_{POIs}.

   $$pois(R_{POIs}) = \{POI_1 \mid (POI_1, POI_2) \in R_{POIs}\} \cup \{POI_2 \mid (POI_1, POI_2) \in R_{POIs}\}$$

   Then, we can define the set of POIs related with a given POI in R_{POIs}.

   $$rel(POI, R_{POIs}) = \{POI' \mid (POI, POI') \in R_{POIs}\} \cup \{POI' \mid (POI', POI) \in R_{POIs}\}$$

   Finally, we need a new trace function that returns a single trace of values that are obtained from all the POIs related with a POI in R_{POIs}.

$$trace\_rel(input, POI, R_{POIs}) =$$
$$[VEF((POI', v, ai)) \mid (POI', v, ai) \in trace(input) \ \wedge \ POI' \in rel(POI, R_{POIs})]$$

We can now define an equality function that is able to deal with replicated POIs.

$$equal(trace_{old}, trace_{new}, R_{POIs}) =$$
$$\bigwedge_{POI \in pois(R_{POIs})} CFUN(trace(input, POI), trace\_rel(input, POI, R_{POIs}))$$

In these approaches, the "equality" function does not operate completely with the "=" operator. Thanks to the flexibility given to the user during the comparison, two traces are considered "equal" when they fit the conditions defined in the comparison function *CFUN*, which is provided by the user.

Equality functions constitute a new parameter of the approach that determines how the traces should be compared. For the comparison of versions using multiple POIs, it is mandatory to provide such a function, while for the test case generation phase depicted in Section 8.3.4 it can be optional. In the test case generation scenario, the user could be interested in obtaining more sophisticated test cases by providing their own equality function. In order to enable this option, an additional parameter can be added to the *tgen* function to provide a customised equality function that should be used when checking whether a trace has been previously computed.

8.5 POI testing in concurrent environments

Non-deterministic computations are one of the main obstacles for regression testing. In fact, they prevent us from comparing the results of a test case executed in different versions because the discrepancies found can be well produced by sources of non-determinism such as concurrency. In some specific situations, however, we can still use POI testing to report whether the behaviour of a concurrent program is preserved. For instance, consider the client-server model depicted in Figure 8.17. In this simple example, a POI should not be placed in *Server*, because we cannot know *a priori* whether req_1 is going to be served before or after req_2, and this could have an impact on the traces obtained from that POI. However, we could place a POI in any of the clients, as long as the request is not affected by the state of the server. This is acceptable for many kinds of server, but it is still a quite annoying limitation for many others.

However, in Erlang, as it is common in other languages, there is a high-level way to define a server. In particular, real Erlang programmers tend to use the Erlang-OTP's behaviour named `gen_server`. By implementing this behaviour, the programmer is only defining the concrete behaviours of a server, leaving all the low-level aspects to the internals of the `gen_server` implementation. These concrete behaviours include how the server's state is initialized, how a concrete request should be served, or what to do when the server is stopped. When using

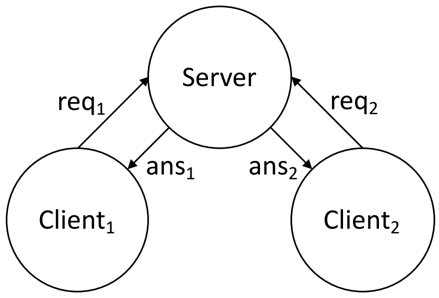


FIGURE 8.17: A simple client-server model

`gen_server`, programmers could use the functions implementing these concrete behaviours as input functions of POI testing. In this way, they can check, for instance, that a server is going to reply the user and leave the server's state in the same way across different versions of the program.

8.6 Implementation

We have implemented the POI Testing approach in a tool named SecEr (Software Evolution Control for Erlang), publicly available at: `https://github.com/mistupv/secer`. SecEr is a tool for Erlang able to automatically generate a test suite that checks the behaviour of a point of interest. It can be used for regression testing, by generating a test suite for a future comparison or by automatically comparing two releases of an Erlang module. SecEr implements the POI testing approach in Erlang. The comparison performed by the tool focuses on a set of program points specified by the user. The tool automatically generates a test suite that checks the behaviour of those program points. These test cases try to maximise the branch coverage, covering a large quantity of different possible execution, and the tool notifies the user of any unexpected result. SecEr is implemented in Erlang and the implementation contains more than 4600 lines of code divided into 11 different Erlang modules with a high level of communication. SecEr is open source and publicly available at:

`https://github.com/mistupv/secer`

Figure 8.18 contains a scheme with the architecture of SecEr. In this figure, rounded squares stand for specific tasks (written in bold inside them) of the POI testing approach, solid squares stand for Erlang modules implemented from scratch, and dashed squares stand for external Erlang modules from existing tools used during the process. Solid arrows represent explicit module calls, dotted arrows new Erlang node creations, and dashed arrows thread creations. The tool is formed by 11 different modules strongly communicated. Each module performs a particular functionality dividing the responsibilities of the POI testing process:

- SecEr. This module is the entrance module when we call SecEr. The module receives and processes the arguments, creates the Erlang nodes where both code versions will be executed in parallel, and create threads

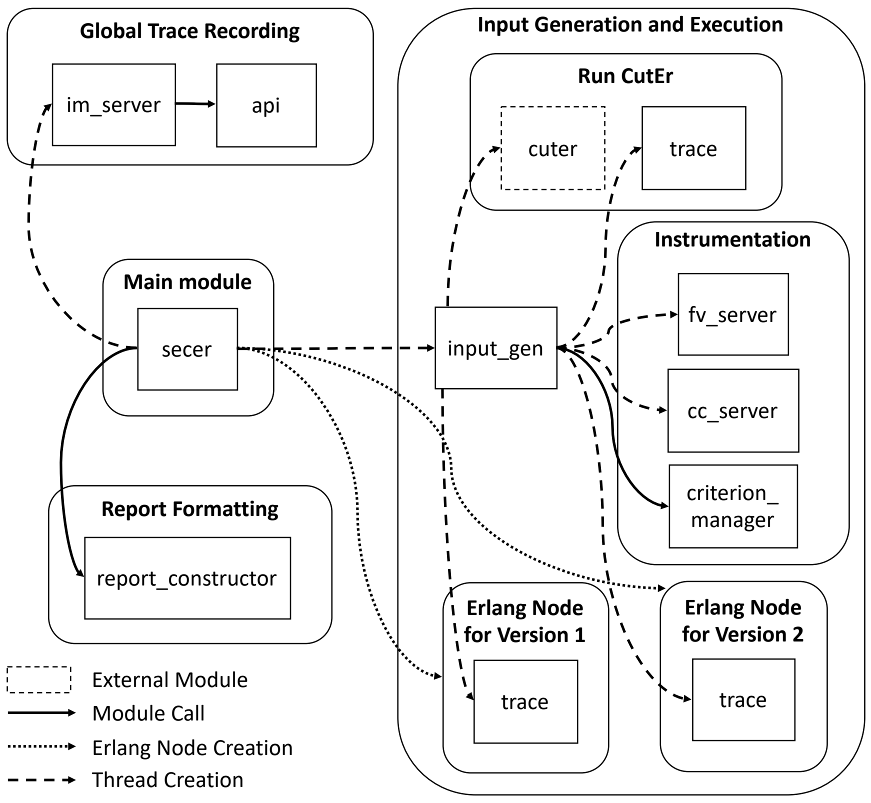


FIGURE 8.18: SecEr modules and the communicate between them to perform POI testing

of two modules: `im_server` and `input_gen`. Finally, it is also in charge of generating the output by performing a call to the `report_constructor` module.

- `im_server`. This module represents the "input manager" server. The server of this module receive different messages from `input_gen` process, and is the module in charge of comparing the traces produced by the two program version executions. The state of this server includes the POI correspondence given by the user, the comparison function defined by the user (a default comparison function extracted from the `api` module if no function is given), and the traces generated so far classified in three different groups: traces considered *different* by the comparison function, traces considered *equal* by the comparison function, and the cases where one (or both) program executions resulted in a *timeout* before generating any trace value.

- `input_gen`. This module is in charge of the input generation, and is the most interconnected module of the tool. First of all it connects the tool with CutEr, obtaining a set of initial inputs (thanks to the `trace` module) with a high branch coverage over programs. Then, it is in charge of the instrumentation of both code versions, where it needs to create threads of the `fv_server` and the `cc_server` processes. After that, the module creates a `trace` thread (one in each Erlang node) for each program version.

Finally, if the desired coverage or the time limit have not been reached yet, the module spend the rest of the time generating more random inputs, executing them, and managing the traces generated by both programs.

- `trace`. The `trace` module records the sequence of values the POI is evaluated to during the execution of a program. During the CutEr execution, this process stores the "hidden" inputs used by the CutEr tool. Contrarily, in the rest of cases it stores the sequence of values (trace) generated by the execution of a program with a particular input. The trace is stored as a tuple that contains the POI, the value, and a map with the additional information specified by the user.

- `fv_server`. This module is the one in charge of feeding the instrumentation process with the "free variables" (variables that are not defined in the current scope) needed to make the transformation of the code. These variables are used by the rules defined in Figure 8.10.

- `cc_server`. The "conversion container" server module represents a server that stores map with information of the program being instrumented. It provides information about the last id used in the AST node labelling, the POI information given by the user to find them at the AST, and a map with the correspondence between each POI and the corresponding AST id.

- `criterion_manager`. This module is the one that performs the real instrumentation. It locates the path from the AST node to the POI and divide this path in two like described in Section 8.3.2. Finally, the module applies the rules in Figure 8.10 to transform the code. It is worth mentioning that, during the process, the module makes use of the services provided by threads `fv_server` and the `cc_server` to obtain the instrumented AST.

- `report_constructor`. This module is called at the end of the process when either the desired coverage of the generated test cases or the timeout are reached. This module expands all the information packaged in the `im_server` process, counting and classifying the detected errors, and summarises all this information in a report with the information specified by the user.

- `api`. The `api` module mainly contains different forms of configuration that can be used by SecEr. These configurations include VEF, TECF, and most of the configuration modes explained in Section 8.2.1. Some of these functions are used by the default SecEr configuration, but all of them can be used by the user when defining her own configuration file.

Additionally, SecEr includes a couple of extra modules that work at metaprogramming level: `smerl` and `typer_mod`. `smerl` extracts the AST of the Erlang program given, while `typer_mod` is a modification of the TypEr module that extracts Erlang types as a result instead of printing them in the standard output.

8.7 Experimental Evaluation

In this section we study the performance and the scalability of SecEr. In particular, we compare different configurations and study their impact on the performance. First, we collected examples from commits where some regression is fixed. Most of the considered programs were extracted from `EDD` [26][13] because this repository contains programs with two code versions: one version of the code with a bug and a second version of the same code with the expected behaviour, i.e. where the bug is fixed. In order to obtain representative measures, the experiments were designed in such a way that each program was executed 21 times with a timeout of 15 seconds each. The first execution was discarded in all cases (because it loads libraries, caches data, etc.). The average computed for the other 20 executions produced one single data. We have repeated this process enabling and disabling the two most relevant features of SecEr: (i) the use of CutEr and (ii) the use of mutation during the ITC generation. The goal of this study is to evaluate how these features affect the accuracy and performance of the tool. To compare the configurations we computed three statistics for each experiment: the average amount of generated tests, the average amount of mismatching tests, and the average percentage of mismatching tests with respect to the generated ones.

	CUTER+MUTATION			**NO CUTER**			**NO MUTATION**		
	Generated	Mismatching	%	Generated	Mismatching	%	Generated	Mismatching	%
ackermann	13.9	12.9	93.274%	**21.8**	**21.8**	**100.0%**	12.85	11.65	91.27%
caesar	37765.94	1615.1	4.2714%	**103072.0**	**4534.95**	**4.3997%**	38830.55	1702.7	4.3865%
complex_number	69420.2	67236.55	96.8549%	**89670.2**	**86891.75**	**96.9015%**	67451.75	65349.95	96.8825%
erlson1	14780.05	1.55	0.0105%	**14966.2**	**2.65**	**0.0177%**	14872.5	1.9	0.0127%
erlson2	15494.5	**0.95**	0.0059%	**16758.59**	0.8	0.0047%	15553.8	**0.95**	**0.0061%**
mergesort	29718.35	25634.45	86.2585%	**34315.1**	**29622.9**	**86.3259%**	29994.3	25884.2	86.299%
rfib	28.05	28.05	**100.0%**	**29.0**	**29.0**	**100.0%**	28.4	28.4	**100.0%**
roman	513.79	101.95	19.8415%	**535.35**	**108.05**	**20.1801%**	512.2	101.7	19.8461%
sum_digits	426.3	422.3	99.0615%	**534.0**	**534.0**	**100.0%**	434.0	430.0	99.078%
ternary	85.9	28.05	29.4187%	**1005.4**	**323.25**	**32.2485%**	130.0	39.7	27.8311%
turing	41828.65	28268.95	67.5825%	**77247.45**	**52651.5**	**68.1595%**	41573.1	28150.95	67.7135%
vigenere	115.55	2.1	1.269%	**308.7**	**4.59**	1.1849%	114.9	1.95	**1.4849%**

Table 8.1: Experimental evaluation of three SecEr configurations with a timeout of 15 seconds

Table 8.1 summarizes the experiments[14], where the best result for each program has been highlighted in bold. These results show that our mutation technique is able to produce better test cases than random test generation. Clearly, the configuration that does not use CutEr (the one in the middle) is almost always the best: it generates in all cases the highest amount of test cases, and it also generates more mismatching test cases (except for the *erlson2* program). The interpretation of these data is the following: CutEr invests much time to obtain the initial set of inputs, but the concolic test cases it produces do not improve enough the quality of the suite. This means that in general it is better to invest that time in generating random test cases, which on average produce more mismatching test cases. There are two exceptions: *erlson2* and *vigenere*.

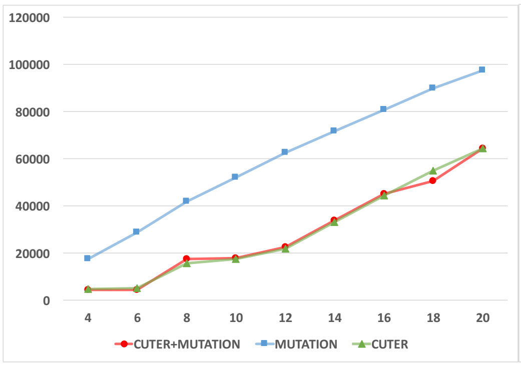
[13]`https://github.com/tamarit/edd/tree/master/examples`

[14]All data and programs used in this experiment are available online at `https://github.com/mistupv/secer/tree/master/benchmarks`

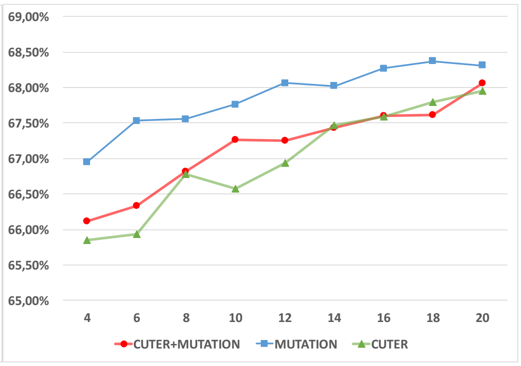
In *erlson2* the error is related to a very particular type of input (less than 0.02% of the generated tests report this error), and CutEr directs the test generation to mismatching tests in a more effective way. With respect to the second program (*vigenere*), although the configuration that does not run CutEr generates more mismatching tests than the rest, the tests generated by CutEr allow the tool to reach a mismatching trace faster. This is the reason for the slight improvement in the mismatching ratio. The common factor in both programs is that the mismatching ratio is rather low. This is a clear indication that CutEr can be useful when some corner cases are involved in the regression.

We can conclude that the results obtained by the tool are strongly related to the location of the error, and on the type of error. If it is located in an infrequently executed code or it is a corner case, the most suitable configuration is the one running CutEr. In contrast, if the error is located in a usually executed code, we can increase the mismatching tests generation by disabling CutEr. Because we do not know beforehand what is the error and where it is, the most effective way of using the tool is the following: First, run SecEr without CutEr, trying to maximize the mismatching test cases. If no discrepancy is reported, then enable CutEr to increase the reliability of the generated test cases.

We have also evaluated the growth rate of the generated test cases and of the percentage of mismatching ITCs. For this experiment we selected the program *turing* because it produces a considerable amount of tests in the three configurations, and also because the mismatching ratio is similar in all of them and not too close to 100%. We ran the three configuration of SecEr with this program with a timeout ranging between 4 and 20 seconds with increments of 2 seconds.



(A) Number of tests generated for `Turing`



(B) Mismatching ratio produced for `Turing`

The results of the experiment are shown in Figures 8.19a and 8.19b. In Figure 8.19a, the X axis represents SecEr timeouts (in seconds), and the Y axis represents the number of test cases generated. In all cases, the configuration that does not run CutEr generates the highest number of tests. This configuration has a linear growth. On the other hand, the configurations using CutEr show a slow onset. They need at least twelve seconds to reach a considerable increase in the number of generated tests. There is not a significative difference between the configurations using CutEr. This means that the mutation technique does not slow down the test generation. In Figure 8.19b, the X axis represents SecEr timeouts (in seconds), and the Y axis represents the percentage of mismatching tests over the total amount of tests generated. Clearly, in the three configurations, the quality of the generated test cases increases over time (i.e., the mismatching tests ratio increases over time). The configuration that does not run CutEr presents the highest percentage. In this case, the two approaches using CutEr produce different results. With smaller timeouts, it is preferable to enable mutation.

8.8 Related Work

First of all, it would be interesting to classify our approach inside the categories defined by previous researches in the test generation context. In the bibliography, there are some surveys and papers about test case generation, like the orchestrated survey of methodologies for automated software test case generation ([8]), or the survey of test amplification [47]. On the one hand, [8] identifies five techniques to automatically generate test cases. In this work, our approach could be included in the class of *adaptive random technique as a variant of random testing*. Inside this class, the authors identify five approaches. Our mutation of test inputs shares some similarities with some of these approaches like selection of best candidate as next test case or exclusion. On the other hand, [47] identifies four categories that classify all the work done in the field. Our work could be included in the category named *amplification by synthesising (new tests with respect to changes)*. Inside this category, our technique falls on the "other approaches" subcategory.

Automated behavioural testing techniques like Soares et al. [184] and Mongiovi [137] are similar to our approach, but they are restricted in the kind of changes that can be analysed (they only focus on refactoring). In contrast, our approach is independent of the kind (or the cause) of the changes, being able to analyse the effects of any change in the code regardless of its structure.

There are several works focused on the regression test case generation. DiffGen [191] instruments programs to add branches in order to find the differences in the behaviour between two versions of the program, then, it explores the branches in both versions, and finally it synthesises test cases that show the detected differences. An improvement of this approach is implemented in the tool eXpress [192] where the irrelevant branches are pruned in order to improve the efficiency. Our technique, in contrast, is not directed by the computation paths, but it is directed by the POIs (i.e., what the user wants to observe). Another related approach is model-based testing for regression testing. This approach studies the changes made in a model, e.g., UML [142] or EFSM models based on dependence analysis [35], and test cases are generated from them. We do not require any input model neither infer it, so although some ideas are similar, the way to implement them is completely different. Finally, there are works that use symbolic execution focused on the regression test generation, like [159] and [27]. All these works are directed to maximise the coverage of the generated test suites. Moreover, they need an existing regression test case suite to start with. There exists alternatives, but with the same foundations, like [214] where a tree-based approach is proposed to achieve high coverage. Our approach is not directed by coverage, but instead by the POIs, and we do not require any regression test as input.

Automated regression test case generation techniques like Korel and Al-Yami [98] are also very similar to our approach, but the user can only select output parameters of a function to check the behaviour preservation. Then, their approach simply runs that specific function and checks whether the produced values are equal for both versions of the program. Therefore, their approach helps to discover errors in a concrete function, but they cannot generate inputs for one function and compare the outputs of another function. This limits, e.g., the observation of recursion. In contrast, we allow to select any input function and place the POIs in any other function of the program. Additionally, their test input generation relies on white-box techniques that are not directed by the changes. Our approach, however, uses a black-box test input generation which is directed by changes.

Yu et al. [211] presented an approach that combines coverage analysis and delta debugging to locate the sources of the regression faults introduced during some software evolution. Their approach is based on the extraction and analysis of traces. Our approach is also based on traces although not only the goals but also the inputs of this process are slightly different. In particular, we do not require the existence of a test suite (it is automatically generated), while they look for the error sources using a previously defined test suite. Similarly, Zhang et al. [213] use mutation injection and classification to identify commits that introduce faults.

The Darwin approach [160] starts from the older version of the program, a program that it is known to be buggy, and an input test case that reveals the bug. With all this information, it generates new inputs that fail on the buggy program, and then run them using dynamic symbolic execution and storing the produced trace (in their context, a trace contains the visited statements). Finally, the traces from the buggy version and from the old version are compared to locate the source of the discrepancy. Although the approach could seem similar to ours, the goals are different. We try to find discrepancies, while they start from an already-found discrepancy.

Our technique for mutation of inputs share some similarities with RANDOOP [149]. In their approach, they start from test cases that do not reveal any failure, and randomly construct more complex test cases. The particularity of their approach is that the random test generation is feedback-directed, in the sense that each generated test case is analyzed to take the next decision in the generation. We do something similar, although our feed-back is directed by the POIs selected by the user.

DSpot [15] is a test augmentation technique for Java projects that creates new tests by introducing modifications in the existing ones. The number of variants that will be generated is known beforehand and determined by parameters like the number of operations or the number of statements. In order to define the output of a test case, they introduce a concept called *observation point*, which is similar to our POIs. The difference is that they define and select their observation points (in particular, attribute getters, the `toString()` method, and the methods inside an assertion) while in our approach is the user who defines them. Additionally, our approach does not need an already existent test suite.

Sieve [164] is a tool that automatically detects variations across different program versions. They run a particular test and store a trace that in their context is a list of memory operations over variables. The generated traces are later studied in order to determine what have changed in the behaviour and why it has changed. Although their goal is not the same as ours, their approach shares various similarities with ours, in particular code instrumentation and trace comparison.

Mirzaaghaei [135] presented a work called *Automatic Test Suite Evolution* where the idea is to repair an existing test suite according to common patterns followed by the practitioners when they repair a test suite. A repair pattern can be something like the introduction of an overloaded method. A modification of our technique could be used to achieve a similar goal, by not only producing test case input, but also repair patterns in order to check whether they are effective repairing an outdated test suite.

There are other approaches that use traces to compare program versions, like the ones based on program spectra [170]. Different program spectra have been proposed (branch, execution trace or data dependence spectra), but value spectra [208] is the most similar to our additional information used to trace method calls. In particular, value trace spectra record the sequence of the user-function executions traversed as a program executes. After the spectra recording, spectra comparison techniques are used to find value spectra differences that expose internal behavioural deviations inside the black box. However, the spectrum is generated for all the user-defined functions while in our approach users decide which functions should be compared. Additionally, POI testing allows a more flexible use of these call traces. Finally, the motivation and also some techniques of the call traces are similar to the ones of *algorithmic debugging* [177]. In fact, this approach has been successfully applied to Erlang [26].

Most of the efforts in regression testing research have been put in the regression testing minimisation, selection, and prioritisation [210], although among practitioners it does not seem to be the most important issue [50]. In fact, in the particular case of the Erlang language, most of the works in the area are focused on this specific task [24, 193, 197, 199]. We can find other works in Erlang that share similar goals but more focused on checking whether applying a refactoring rule will yield to a semantics-preserving new code [95, 115].

With respect to tracing, there are multiple approximations similar to ours. In Erlang's standard libraries, there are implemented two tracing modules. Both are able to trace the function calls and the process related events (spawn, send, receive, etc.). One of these modules is oriented to trace the processes of a single Erlang node [51], allowing for the definition of filters to function calls, e.g., with names of the function to be traced. The second module is oriented to distributed system tracing [53] and the output trace of all the nodes can be formatted in many different ways. Cronqvist [45] presented a tool named redbug where a call stack trace is added to the function call tracing, making possible to trace both the result and the call stack. Till [194] implemented erlyberly, a debugging tool with a Java GUI able to trace the previously defined features (calls, messages, etc.) but also giving the possibility to add breakpoints and trace other features such as exceptions thrown or incomplete calls. All these tools are accurate to trace specific features of the program, but none of them is able to trace the value of an arbitrary point of the program. In our approach, we can trace both the already defined features and also a point of the program regardless of its position.

Chapter 9

Design-by-contract verification in Erlang

Defensive programming is a way to eliminate unexpected behaviours, e.g., division by zero, by checking the validity of arguments before operations are attempted. However, in mainstream programming this style is not a recommendable practice for various reasons, such as the need to add *boiler-plate* code which obscures the program logic, and because of the execution time overhead caused by such checks. In fact, the language considered in this chapter, Erlang, is infamous for its stance that defensive programming is to be avoided – "let it crash".

Erlang programming has a number of interesting and innovative mechanisms for error detection and recovery,[1] but these features are supposed to be used only for errors that are hard to avoid (i.e., because static type systems are not strong enough to characterise full program behaviour). Unfortunately, not all the errors fall in this category. Some errors are rather easy to detect, and should ideally be detected at "compile time" instead of being detected and corrected when a software is operational. Erlang, as Python or JavaScript, is a dynamic programming language. This means that the program compiler, if it even exists, does relatively few checks during compilation to help prevent errors when the program is later run. For this reason, static analysis techniques, popularised primarily by the Dialyzer tool [120], have been successfully and widely adopted by Erlang practitioners. Dialyzer can analyse the code and report some errors without requiring annotating program code in any way. However, the capability of Dialyzer to detect program bugs can be considerably improved by the use of type contracts [94]. Note that such contracts are not used only by Dialyzer to implement static type checking, but also serve to document the developed code, which improves the maintainability of the resulting software.

Even when the Dialyzer tool is used, and when programmers provide e.g. EUnit test cases (an Erlang testing tool) to further check program behaviour, program bugs can still remain. In this work, we propose a mechanism to further structure and strengthen such "defensive" programming tasks, i.e., the Erlang Design-By-Contract (EDBC) system, a runtime verification framework based on the Design-By-Contract [134] philosophy. The EDBC system is available as free and open-source software at `https://github.com/serperu/edbc`.

[1]For example, process links help structure fault detection and fault recovery in complex applications.

In typical design-by-contract frameworks there are different types of contracts, with most of them being related to program functions or methods. The most common contracts are *pre-* and *postconditions* (see Chapter 7, Section 7.2). In addition to these, the EDBC system includes contracts to control other aspects of functions and methods: *type contracts*, to check the correct dynamic typing given by Erlang, *decreasing-argument contracts* to help analyse program termination, *execution-time contracts* to document bounds for the execution time of functions, and *purity contracts* which prohibit side effects such as Erlang process-to-process communication. All these contracts can be used in any Erlang program, regardless of whether the program is purely functional, or structured as a concurrent system, composed of a number of concurrent processes. To avoid the traditional execution overhead associated with the use of contracts, normally they are checked only during software production and maintenance. Consequently, the EDBC library provides a mechanism to disable such checks for running product code.

The rest of the chapter illustrates with examples how to use both sequential and concurrent contracts in the implemented approach (Section 9.1). Finally, the architecture and some implementation details are described (Section 9.2), such as the transformation processes used to correctly check the defined contracts in the correct order.

9.1 Contracts in Erlang

In this section, we introduce the contracts provided by the EDBC system and show how they can be used to check program behaviour at runtime with examples. The contracts proposed in EDBC are divided into two groups: contracts applicable in sequential Erlang, and contracts only applicable to concurrent Erlang. It is worth mentioning that, as an implementation decision, we have chosen to use Erlang macros to represent the syntax of all contracts. The reason is that similar tools like EUnit also uses macros for assert definitions.

9.1.1 Contracts for sequential Erlang

First of all, it is important to remark that the contracts described here are not only applicable in sequential Erlang programs, but they can be used in both sequential and concurrent programming. The contracts for sequential Erlang code provide additional information about Erlang functions. They can be used to document the way an Erlang function may be called or to delimit the behaviour that can be expected when calling it (i.e., concerning the computed result, side effects, termination properties, and so on).

1. **Precondition contracts.** With the macro `?PRE/1` we can define a precondition that a function should hold. The macro should be placed before the first clause of the annotated function. The single argument of this macro is a function without parameters, e.g. `fun pre/0` or an anonymous function `fun() -> ... end`, that we call *precondition function*. A precondition function is a plain Erlang function. Its only particularity is

that it includes references to the function parameters. In Erlang, a function can have more than one clause, so referring the parameter using the user-defined names can be confusing for both EDBC and for the user. In order to avoid these problems, in EDBC we refer to the parameters by their position. Additionally, the parameters are rarely single variables but can be more complex terms like a list or a tuple (since Erlang permits pattern matching). For these reasons we use the EDBC's macro `?P/1` to refer to the parameters. The argument of this macro should be a number that ranges from 1 to the arity of the annotated function. For instance, `?P(2)` refers to the second parameter of the function. A precondition function should return a boolean value which indicates whether the precondition of the annotated function holds or not. The precondition is checked during runtime before actually performing the call. If the precondition function evaluates to `false`, the call is not preformed and a runtime exception is raised.

Example 9.1. *Consider a function* `find(L,K)` *that searches for the position of a value* `K` *in a list* `L`*, and returns* `-1` *if the value is not found. Figure 9.1a shows the usage of a precondition contract which expresses that the first parameter list should not be empty.*

```
?PRE(fun() -> length(?P(1)) > 0 end).
find(L, K) -> ...
```

(A) Erlang function `find/2` annotated with contracts

```
** exception error: {"Precondition does not hold.
Call find([], 3), the list is empty."}
```

(B) Precondition contract violation

FIGURE 9.1: Precondition Contract

In case the precondition is violated, an Erlang exception is raised. For instance, if we attempt the function call `find([], 3)`*, which fails the precondition check because the length of the list argument is 0, the resulting error message is shown in Figure 9.1b.*

2. **Postcondition contracts.** Similar to preconditions, the macro `?POST/1` is used to define a postcondition that a function should satisfy. The macro should be placed after the last clause of the annotated function. Its argument is a function without parameters, which we call the *postcondition function*. In order to check postconditions, a way to refer to the result of the function is essential. For this reason, we have defined the macro `?R`, that refers to the result of the function call. Additionally, as in the `?PRE/1` precondition function, the `?P/1` macros can be used to refer to the actual parameters of the annotated function. The result of a postcondition function is also a boolean value. Postcondition functions are checked after a call terminates, and a runtime error is raised if the postcondition function evaluates to `false`.

Example 9.2. *Figure 9.2a shows an example of a postcondition contract associated with the function* ***find/2****. In this contract, an error is raised if the index returned by* ***find/2*** *is greater that the length of the list.*

```
find(L, K) -> ...
?POST(fun() -> ?R < 0 orelse
               ?R < length(?P(1)) end).
```

(A) Erlang function `find/2` with contract annotations

```
** exception error: {"Postcondition does not hold.
Call find([1,2,3], 3), returned value 5."}
```

(B) Postcondition contract violation

FIGURE 9.2: Postcondition Contract

Suppose an implementation of ***find/2*** *returns the value* ***5*** *to the call* ***find([1,2,3],3)****. In this case, the execution would raise the error shown in Figure 9.2b.*

3. **Decreasing-argument contracts.** These contracts are meant to be used in recursive functions, and check that (some) arguments are always decreasing in nested calls[2]. There are two types of macros to define these contracts: `?DECREASE/1` and `?SDECREASE/1`. They both operate exactly in the same way with the exception that the `?SDECREASE/1` macro indicates that the argument should be strictly smaller in each nested call, while the `?DECREASE/1` macro also permits the argument to be equal. The argument of both macros can be either a single `?P/1` macro or a list containing several `?P/1` macros. These contracts should be placed before the first clause of the function. Decreasing-argument contracts are checked each time a recursive function call is made, by comparing the arguments of the current call with the nested call just before performing the actual nested recursive call. In case the argument expected to decrease is not actually decreasing, a runtime error is raised and the call is not performed.

   Example 9.3. *The functionality of this contract is shown in Figure 9.3. We use a wrong implementation of the Erlang program calculating the Fibonacci numbers shown (Figure 9.3a). When executing the function call* ***fib(2)****, the error message in Figure 9.3b is shown.*

4. **Execution-time contracts.** EDBC introduces two macros that allow users to define contracts concerning execution times: `?EXPECTED_TIME/1` and `?TIMEOUT/1`. The macros should be placed before the first clause of the annotated function. The argument of these macros is a function without parameters called the *execution-time function*. An execution-time function should evaluate to an integer which defines the expected execution time in milliseconds. Within the body of an execution-time

[2]Note that decreasing contracts only guarantee termination if the sequence is strictly decreasing and well founded, i.e. values cannot go below a certain limit.

```
?SDECREASES(?P(1)).
-spec fib(integer()) -> integer().
fib(0) -> 0;
fib(1) -> 1;
fib(N) -> fib(N - 1) + fib(N + 2).
```

(A) Erlang function `fib/1` annotated

```
** exception error: {"Decreasing condition does
not hold. Previous call: fib(2).
Current call: fib(4).", [{ex,fib,1,[]},...
```

(B) Decrease contract violation

Figure 9.3: Decrease Argument Contract

function we can use `?P/1` macros to refer to the arguments. Permitting the execution-time function to refer to arguments is particularly useful when dealing with lists or similar structures where the expected execution time of the function is related to the sizes of arguments. Both macros have a similar semantics, the only difference is that with macro `?EXPECTED_TIME/1` the EDBC system waits till the evaluation of the call finishes to check whether the call takes the expected time, while with macro `?TIMEOUT/1` EDBC raises an exception if the function call does not terminate before the timeout limit is reached. As an example of time contracts, we consider a function which performs a list of tasks. Each task has its type (even or odd), and the allowed execution time is defined by this type (100 and 200 ms., respectively).

Example 9.4. *Figure 9.4a shows the function and its associated time contract. Supposing we change the execution-time function to, for instance,* `fun() -> 20 + (length(?P(1)) * 100) end`*, we would obtain the contract-violation report shown in Figure 9.4b.*

```
?EXPECTED_TIME(fun() ->
    20 + lists:sum([case (I rem 2) of
    0 -> 100; 1 -> 200 end || I <- ?P(1)]) end)
f_time(L) -> [f_time_run(E) || E <- L].
f_time_run(N) when (N rem 2) == 0 -> timer:sleep(100);
f_time_run(N) when (N rem 2) /= 0 -> timer:sleep(200).
```

(A) Function with Execution-time contracts

```
** exception error: {"The execution of
ex:f_time2([1,2,3,4,5,6,7,8,9,10]) took too
much time. Real: 1509.913 ms.
Expected: 1020 ms. Difference: 489.913 ms)
```

(B) Execution-time contract violation

Figure 9.4: Execution-time Contract

5. **Purity contracts.** When we say that a function is pure we mean that its execution does not cause any side effects, i.e., it does not perform I/O

operations, nor does it send messages, etc. That a function is "pure" can be declared by using the macro `?PURE/0` before its first clause. The purity checking process is performed in two steps. First, before a call to an function declared to be pure is performed, a tracing process is started. Then, once the evaluation of the annotated function call finishes, the trace is inspected. If a call to an impure function or operation has been made, a runtime exception is raised. Note that due to the use of tracing we can provide exact purity checks, ensuring that there are neither false positives nor false negative reports. Note that purity checking is not compatible with execution-time contracts, since checking execution times do require performing impure actions.

Example 9.5. *In order to illustrate the checking of purity contracts we take a simple example used to present* `PURITY` *[155], i.e., an analysis that statically decides whether a function is pure or not. The example the authors presented is depicted in Figure 9.5a. We only added the contract* `?PURE` *in the test case* `g4/0`*, because the other test case, i.e.* `g3/0`*, performs the impure operation* `erlang:put/2`*. When* `g4/0` *is run, no contract violation is reported as expected.*

```
fold1(Fun, Acc, Lst) -> lists:fold(Fun, Acc, Lst).
fold2(Lst, Fun) -> fold1(Fun, 1, Lst).
g3() -> fold1(fun erlang:put/2, ok,
                [computer, error]).
?PURE.
g4() -> fold2([2, 3, 7], fun erlang:'*'/2).
```

(A) Example taken from `PURITY` [155]

```
** exception error: {"The function is not pure.
Last call: ex:g3().
It has call the impure BIF erlang:put/2
when evaluating g3().",[{ex,g3,0,[]}]}
```

(B) Pure contract violation

FIGURE 9.5: Purity Contract

On the other hand, in case we add the contract `?PURE` *to* `g3/0`*, then the execution will fail showing the error report in Figure 9.5b.*

6. **Type contracts.** Erlang has a dynamic type system, i.e., types are not checked during compilation but rather at runtime. However, the language still permits to specify type contracts (represented by `spec` attributes) which serves both as code documentation, and as aid to static analysers like `Dialyzer` [120]. However, such type contracts are not checked at runtime by the Erlang interpreter, because of the potential associated cost at execution time. However, for programs still in production, checking such type contracts during runtime can be helpful to detect unexpected behaviours. For this reason, before a function is evaluated, EDBC checks the type contract of its parameters (if any), while its result is checked

after its evaluation. If a type error is detected, a runtime exception error is raised. Note that EDBC does not use any special macro to check type contracts, the standard `spec` attributes are used instead. Figure 9.6 shows an error that would be shown in case of calling `fib(a)` for the program defined in Figure 9.3b.

```
** exception error: {"The spec precondition does not hold.
Last call: ex:fib(a).
The value a is not of type integer().", ...}
```

FIGURE 9.6: `spec` contract-violation report

Note that the EDBC system can be used to define quite advanced contracts. As a comparison point, the `Dafny` tool [114], which was an inspiration for EDBC, permits the use of quantifiers to define conditions for input lists. Figure 9.7a shows as an example of how quantifiers are used in `Dafny` to characterise the function `Find/2`.

```
method Find(a: array<int>, key: int)
  returns (index: int)
  requires a != null
  ensures 0 <= index ==> index < a.Length &&
         a[index] == key
  ensures index < 0 ==> forall k ::
    0 <= k < a.Length ==> a[k] != key
{...}
```

(A) Function `Find/2` annotated in `Dafny`

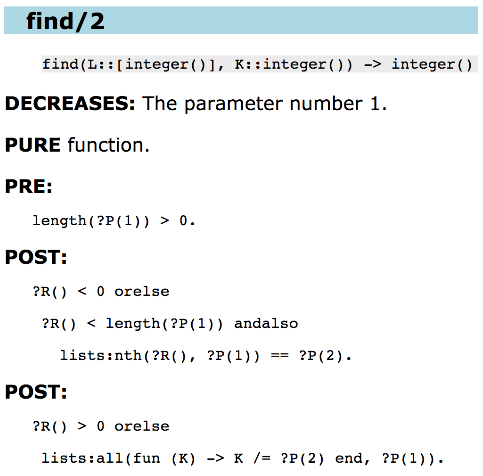
```
?PRE(fun() -> length(?P(1)) > 0 end).
?SDECREASES(?P(1))
find(L, K) -> ...
?POST(fun() -> ?R < 0 orelse
       (?R < length(?P(1))
        andalso lists:nth(?R, ?P(1)) == ?P(2))
     end).
?POST(fun() -> ?R > 0 orelse
       lists:all(fun(K) -> K /= ?P(2) end,
       ?P(1))
     end).
```

(B) Function `find/2` annotated in Erlang/EDBC.

FIGURE 9.7: Contracts for function `find/2` annotated in `Dafny` and `Erlang`

Such contracts with quantifiers can be represented in EDBC too. Instead of using a special syntax as in `Dafny`, we can check conditions with quantifiers using a common Erlang function such as `lists:all/2`, which checks whether a given predicate is true for all the elements of a given list. Figure 9.7b shows how the contracts in Figure 9.7a are represented in EDBC. If we implemented this function as a recursive one, the list would be decreasing between calls. Then, we could also add the contract ?SDECREASE(?P(1)) to the function.

Contracts added by users can also be used to generate documentation. Erlang OTP includes the tool EDoc [52] which generates documentation for modules in HTML format.



find/2

```
find(L::[integer()], K::integer()) -> integer()
```

DECREASES: The parameter number 1.

PURE function.

PRE:

```
length(?P(1)) > 0.
```

POST:

```
?R() < 0 orelse
 ?R() < length(?P(1)) andalso
   lists:nth(?R(), ?P(1)) == ?P(2).
```

POST:

```
?R() > 0 orelse
 lists:all(fun (K) -> K /= ?P(2) end, ?P(1)).
```

FIGURE 9.8: EDoc for the annotated function find/2.

We have modified the generation of HTML documents to also include information concerning EDBC contracts. As an example, the EDoc-generated documentation for the function find/2, with information of its contracts (some in Figure 9.7b and some new), and its type specification, is depicted in Figure 9.8. Finally, it is important to note that the contract checking performed by EDBC does not cause incompatibilities with other Erlang analysis tools. For instance, users can both define EDBC contracts, and include EUnit [28] test case assertions, in the same function.

9.1.2 Contracts for concurrent Erlang

The scenarios given in concurrent Erlang code are slightly different from the ones given in sequential Erlang. Normally, concurrent Erlang code is already highly structured. Most Erlang programmers do not write concurrent code from scratch, but rather rely heavily on proven concurrent Erlang behaviours (which can be considered a form of *design patterns*) present in the Erlang/OTP standard library.

The Erlang behaviours are formalisations of common programming patterns. The idea is to divide the code for a process in a generic part (a behaviour Erlang module), which is never changed, and a specific part (a callback Erlang module), which is used to tailor the process for the particular application being implemented. Thus, the behaviour module forms part of the Erlang/OTP standard library, and the callback module is implemented by the programmer. Such behaviours provide standard mechanisms to implement concurrent behaviours: the generic part provides a proven implementation of an often complex concurrent coordination task, which allows programmers to focus on the easier task on how to adapt this generic behaviour to the particular application at hand.

Thus, the definition and use of such behaviours, can be seen as a form of parametric software contract. The generic part of the contract guarantees the general conduct of the behaviour, which to be able to operate correctly, requires the specific part of the contract, i.e., that callback functions work correctly (e.g. are computationally efficient, and terminate normally).

The concurrent part of EDBC focus on an Erlang behaviour, which is heavily used in industrial applications, the `gen_server` behaviour. This behaviour is used to implement client-server architectures. In practice, the behaviour has a number of shortcomings which are addressed in EDBC by extending it to handle client requests which arrive when the server is not capable of servicing them, a common situation in asynchronous concurrent systems. A programmer using the normal `gen_server` behaviour must manually handle such asynchronous requests by implementing a queuing policy in the specific behaviour part. In EDBC, a modified `gen_server` is provided instead, where the generic behaviour part handles such asynchronous requests according to a simple rule, and which implements a flexible queuing policy, thus providing a concurrent contract which is considerably easier to use.

Before describing the concurrent contracts of EDBC, there is another type of contract in EDBC that does not consider the concurrency scenarios but is strictly related to concurrent environments, since it apples restrictions at process level. These contracts are the *invariant contracts*. Invariants are meant to be used in Erlang behaviours which has an internal state. An invariant contract is defined by using the macro `?INVARIANT/1`. This macro can be placed anywhere inside the module implementing the behaviour. The argument of the `?INVARIANT/1` macro is a function, named *invariant function*, with only one parameter that represents the current state of the behaviour. Then, an invariant function should evaluate to a boolean value which indicates whether the given state satisfies the invariant or not. The invariant function is used to check the invariant contract each time a call to a function which is permitted to change the state finishes, e.g., when a call by the `gen_server` behaviour to the `handle_call/3` callback function finishes. Note that invariant contracts can be used to check for the absence of problems in concurrent systems such as e.g. starvation. Examples of invariant contracts and their associated violation report are presented later in Figures 9.12 and 9.13.

The `gen_server` behaviour with contracts

One of the more commonly used Erlang behaviours is the `gen_server`, which provides a standard way to implement a client–server architecture. A callback module for this behaviour needs to define the specific parts of the server (process), e.g., what is the initial state of the server (e.g., implementing the Erlang function `init/0`), and handling specific client requests (e.g., implementing the Erlang functions `handle_call/3` for blocking client calls, and `handle_cast/2` for non-blocking client calls), etc.

Given the highly regular nature of specific parts of this behaviour, the use of some contracts is highly useful. For instance, invariants (`?INVARIANT/1` contracts) constrain the (persistent) state of the underlying server process. Moreover, for the server to function correctly, the generic parts of the service require the specific parts of the behaviour to satisfy a number of properties expressible as contracts: calls to `handle_call/3` (or `handle_cast`) must normally be side effect free (as the generic part handles replying to server requests) as expressed by the `?PURE/0` contract, and the code implementing `handle_call/3` should be efficient as expressed by the `?EXPECTED_TIME/1` contract.

We have also extended the `gen_server` behaviour to handle asynchronous client requests to the server. This extension provides a mechanism for the server to postpone requests which it is not yet ready to serve, but which should be served in the future when the server state changes. Concretely we add a new callback function that the behaviour specific part may implement: `cpre/3`. This function should return a boolean value indicating whether the server is ready or not to serve a given request. The rest of the `gen_server` callbacks are not modified. The three parameters of the callback function `cpre/3` are (i) the request, (ii) the *from of the request*[3], and (iii) the current server state. The function `cpre/3` should evaluate to a tuple with two elements. The first element of the tuple is a boolean value which indicates if the given request can be served. The second element is the new server state.

The `gen_server_cpre` behaviour behaves in the same way as the `gen_server` behaviour except with a significant difference. Each time the server receives a client request, it calls to `cpre/3` callback before calling the actual `gen_server` callback, i.e., `handle_call/3`. Then, according to the value of the first element of the tuple that `cpre/3` returns, either the request is actually performed (when the value is `true`) or it is queued to be served later (when the value is `false`). In both cases, the server state is updated with the value returned in the second element of the tuple.

EDBC includes two implementations of the `gen_server_cpre` behaviour, each one treats the queued requests in a different way. The least complicated implementation resends to itself a client request that cannot be served in the current server state, i.e., a request for which function `cpre/3` returns `{false, ...}`. Since mailboxes in Erlang are ordered according to the arrival time of messages (i.e., in FIFO order), the postponed request will be the last request in the queue of incoming requests. This can be considered unfair, because, when once in the future the state of the server has changed thus potentially permitting the postponed client request to be served, the server could instead serve serve new client requests that have arrived later than the postponed client request.

The EDBC framework also provides a different version (which is fairer) of the `gen_server_cpre` behaviour. In this version, three queues are used to ensure that older requests are served first: $\texttt{queue}_{\texttt{current}}$, $\texttt{queue}_{\texttt{old}}$, and $\texttt{queue}_{\texttt{new}}$. Each time the server is ready to listen for new requests, the $\texttt{queue}_{\texttt{current}}$ is inspected. If it is empty, then the server proceeds as usual, i.e., by receiving a request from its mailbox. Otherwise, if it is not empty, a request from $\texttt{queue}_{\texttt{current}}$ is served. Consequently, the served request is removed from $\texttt{queue}_{\texttt{current}}$. The queues are also modified by adding requests to $\texttt{queue}_{\texttt{old}}$ and $\texttt{queue}_{\texttt{new}}$. This is done when function `cpre/3` returns `{false, ...}`. Depending on the origin of the request

[3]The *from of the request* has the same form as in the `handle_call/3` callback, i.e., a tuple `{Pid,Tag}`, where `Pid` is the process identifier of the client issuing the request, and `Tag` is an unique tag.

it is added to $\texttt{queue}_{\texttt{old}}$ (when it comes from $\texttt{queue}_{\texttt{current}}$) or to $\texttt{queue}_{\texttt{new}}$ (when it comes from the mailbox). Finally, each time a request is completely served, the server state could have been modified. A modification in the server state can enable postponed requests to be served. Therefore, each time the server state is modified, $\texttt{queue}_{\texttt{current}}$ is rebuilt as follows:

$$\texttt{queue}_{\texttt{old}} + \texttt{queue}_{\texttt{current}} + \texttt{queue}_{\texttt{new}}$$

Real scenarios to use `gen_server_cpre` behaviour

We show two real examples where `gen_server_cpre` has been proved useful. Those examples are the *selective receives* scenario and the *readers-writers* problem.

- **Selective receives.** In public forums such as *stackoverflow*[4] and the *erlang-questions* mailing list[5], where Erlang programming is discussed, there have been a number of questions regarding the limitations of the standard `gen_server` implementation. Most of them concern how to implement a server which has the ability to delay some requests. For example, one question posted in stackoverflow.com[6] asks whether it is possible to implement a server which performs a selective receive while using a `gen_server` behaviour. None of the provided answers is giving an easy solution. Some of them suggest that the questioner should not use a `gen_server` for this, and directly implement a *low-level* selective receive. Other answers propose to use `gen_server` but delay the requests *manually*. This solution involves storing the request in the server state and returning a `no_reply` in the `handle_call/3`. Then, the request should be revised continually, until it can be served and use `gen_server:reply/2` to inform the client of the result. Our solution is closer to the last one, but all the management of the delayed requests is completely transparent to the user.

```
handle_call(test, _From, _State) ->
  List = [0,1,2,3,4,5,6,7,8,9],
  lists:map(fun(N) -> spawn(fun() ->
      gen_server:call(?MODULE, {result, N}) end)
    end, lists:reverse(List)),
  {reply, ok, List};
handle_call({result, N}, _From, [N|R]) ->
  io:format("result: " ++ integer_to_list(N) ++ "~n"),
  {reply, ok, R}.
```

FIGURE 9.9: `handle_call/2` for selective receive

Figure 9.9 shows the function `handle_call/2` of the `gen_server` that the questioner provided to exemplify the problem. When the request

[4] https://stackoverflow.com

[5] http://erlang.org/mailman/listinfo/erlang-questions

[6] https://stackoverflow.com/questions/1290427/how-do-you-do-selective-receives-in-gen-servers

`test` is served, it builds ten processes, each one performing a `{result, N}` request, with `N` ranging from 0 to 9. Additionally, the server state is defined as a list which also ranges from 0 to 9 (Figure 9.9, lines 2 and 6). The interesting part of the problem is how the `{result, N}` requests need to be served. The idea of the questioner is that the server should process the requests in the order defined by the state. For instance, the request `{result, 0}` can only be served when the head of the state's list is also `0`. However, there is a problem in this approach. The questioner explains it with the sentence: *when none of the callback function clauses match a message, rather than putting the message back in the mailbox, it errors out.* Although this is the normal and the expected behaviour of a `gen_server`, the questioner thinks that some easy alternative should exists. However, as explained above, the solutions proposed in the thread are not satisfactory enough.

```
cpre(test, _, State) -> {true, State};
cpre({result, N}, _, [N|R]) -> {true, [N|R]};
cpre({result, N}, _, State) -> {false, State}.
```

FIGURE 9.10: `cpre/3` for selective receive

With the enhanced versions of the `gen_server` behaviour we propose in this work, users can define conditions for each request by using function `cpre/3`. Figure 9.10 depicts a definition of the function `cpre/3` that solves the questioner's problem without needing to redefine function `handle_call/3` of Figure 9.9. The first clause indicates to the `gen_server_cpre` server that the request `test` can be served always. In contrast, `{result, N}` requests only can be served when `N` coincides with the first element of the server's state.

- **Readers-writers.** In this example we define a simple server that implements the readers-writers problem, as a second example of the use of the extended `gen_server_cpre` contract. We start by introducing an implementation of the problem using the standard `gen_server` behaviour. The server state is a record defined as:

```
-record(state, {readers = 0, writer = false}).
```

The requests that it can handle are four: `request_read`, `request_write`, `finish_read`, and `finish_write`. The first two requests are blocking (because clients need to wait for a confirmation) while the latter two do not block the client (clients do not need confirmation). Figure 9.11 shows the handlers for these requests. They basically increase/decrease the counter `readers` or switch on/off the flag `writer`.

Having defined all these components, we can already run the readers-writer server. It will start serving requests successfully without any noticeable issue. However, the result in the shared resource is a mess,

```
handle_call(request_read, _, State) ->
  NState = State#state{readers = State#state.readers + 1},
  {reply, pass, NState};
handle_call(request_write, _, State) ->
  NState = State#state{writer = true}},
  {reply, pass, NState}.

handle_cast(finish_read, State) ->
  NState = State#state{readers = State#state.readers - 1},
  {noreply, NState};
handle_cast(finish_write, State) ->
  NState = State#state{writer = false},
  {noreply, NState}.
```

FIGURE 9.11: Readers-writers request handlers

mainly because we are forgetting an important problem: its invariant, i.e. $\neg writer \vee readers = 0$.

```
1 ?INVARIANT(fun invariant/1).
2
3 invariant(#state{ readers = Readers, writer = Writer}) ->
4     is_integer(Readers) andalso Readers >= 0
5     andalso is_boolean(Writer)
6     andalso ((not Writer) orelse Readers == 0).
```

FIGURE 9.12: Readers-writers invariant definition

We can define an invariant for the readers-writers server by using the macro `?INVARIANT/1`. Figure 9.12 shows how the macro is used and the helper function which actually checks the invariant. Apart from the standard invariant (line 6 in Figure 9.12), the function also checks that the state field `readers` is a positive integer and that the state field `writer` is a boolean value (lines 4 and 5).

If we run the server with the invariant defined, we obtain feedback on whether the server is behaving as expected. In this case, the server is clearly not a correct implementation of the problem. Therefore, an error should be raised due to the violation of the invariant. An example of the errors is shown in Figure 9.13.

The error is indicating that the server state was `{state,0,true}` when the server processed a `request_read` which led to the new state `{state,1, true}` which clearly violates the defined invariant. The information provided by the error report can be improved by returning a tuple `{false, Reason}` in the invariant function, where `Reason` is a string to be shown in this contract-violation report after the generic message.

In order to correctly implement this feature, we use the function `cpre/3` to control when a request can be served or not. Figure 9.14 shows a function `cpre/3` which makes the server's behaviour correct and avoids

```
=ERROR REPORT====
** Generic server readers_writers terminating
** Last message in was request_read
** When Server state == {state,0,true}
** Reason for termination ==
** {{"The invariant does not hold.",Last call: readers_writers:handle_call(
    request_read, ..., {state,0,true}). Result: {reply, pass,{state,1,true}}
        ",
    [{readers_writers,handle_call,3,...},...]}, ...}
```

FIGURE 9.13: Failing invariant report

violations of the invariant. It enables `request_read` requests as long as the flag `writer` is switched off. Similarly, `request_write` requests also require the flag `writer` to be switched off and the counter `readers` to be 0. If we rerun now the server, no invariant violation errors will be raised.

```
cpre(request_read, _, State = #state{writer = false}) ->
  {true, State};
cpre(request_read, _, State) ->
  {false, State};
cpre(request_write, _,
    State = #state{writer = false, readers = 0}) ->
  {true, State};
cpre(request_write, _, State) ->
  {false, State}.
```

FIGURE 9.14: Readers-writers `cpre/3` definition

Although this implementation is already correct, it is unfair for writers as they have less chances to access the shared resource. The EDBC code repository[7] includes a number of implementations of the example, which implement various fairness criteria.

9.2 Implementation

EDBC is an Erlang library composed of a set of 8 Erlang modules and 4 shell scripts with more than 4300 lines of code. EDBC allows users to include contracts in their Erlang programs. In this section we describe the architecture of EDBC: the modules it is formed by, the scripts contained in EDBC, the tasks they perform, and the relationships between them. Additionally, we explain how the code is instrumented to support contract checking at runtime. EDBC is implemented in Erlang and is open source and publicly available in:

https://github.com/serperu/edbc

[7]https://github.com/serperu/edbc/tree/master/examples/readers_writers

9.2.1 Architecture

Figure 9.15 shows how the modules of EDBC are classified according to their functionality. In the figure, rounded squares stand for different parts of the EDBC library (written in bold inside them), and solid squares stand for Erlang modules and scripts implemented from scratch. Solid arrows represent explicit module calls, dashed arrows represent input/output Erlang files, and dotted arrows represent behaviour usages.

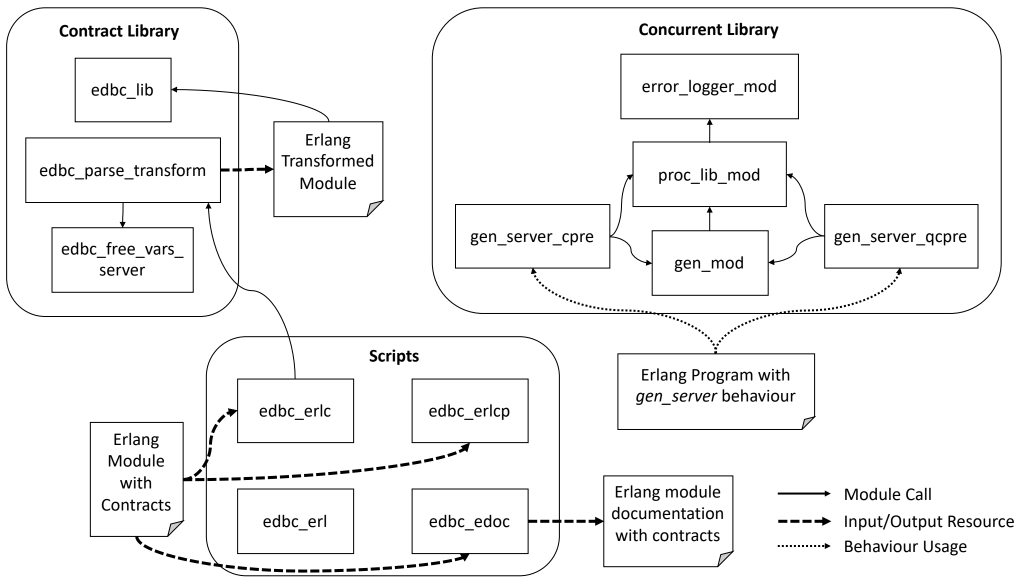


FIGURE 9.15: Architecture of EDBC classified by functionality

- `edbc_parse_transform`. This module is the one that implements the instrumentation of contracts. It finds the macros defining contracts inside the given modules and expands them with new functions that verify each contract without changing the semantics of the program. This module creates a thread of running the `edbc_free_vars_server` module to extract free variables during the instrumentation process, and include in the transformed program successive calls to `edbc_lib`, which will evaluate the contracts during execution.

- `edbc_lib`. This module implements the generic evaluation of all kind contracts. Each function is in charge of evaluating a particular type contract. The inputs of these functions are, in turn, other high order functions that are called inside their body to obtain the results needed to grant the condition in the contract or to report any error found.

- `edbc_free_vars_server`. This module creates a server that generates variable names which are not used in the module being analysed. Each new variable name is formed by adding an identifier character (which is iteratively increased) at the end of the longest-named variable of the original program.

- `gen_server_cpre`. This module defines an enhanced Erlang behaviour based in the `gen_server` behaviour. This module defines an extra callback called `cpre` that represents the condition that needs to be fulfilled by the server to serve a particular request. When the request cannot be served, the message is resent to the server mail list to retry its processing later.

- `gen_server_qcpre`. This module further enhances the `gen_server_cpre` Erlang behaviour. Apart from including the `cpre` callback into `gen_server`, it also includes a queue as a parameter, allowing the customisation of the server requests that cannot be served. This queue can be used to give priority to specific requests types when they are received by the server.

- `gen_mod`. This module is a snapshot of a particular version of the `gen` module of Erlang/OTP that ensures the correct working of `gen_server_cpre` and `gen_server_qcpre`. This module isolates the functionality of this Erlang version and avoids the incompatibility due to changes in the implementation or exportations of the methods in the `gen` module. In the module, all the calls to `proc_lib` module are replaced to calls to the `proc_lib_mod` module[8].

- `proc_lib_mod`. This module is a snapshot of a particular Erlang/OTP module called `proc_lib` with some differences. The `proc_lib_mod` module changes the return values of some methods and replaces all the calls to `error_logger` module by calls to `error_logger_mod`.

- `error_logger_mod`. This module is a snapshot of a particular version of the `error_logger` module in Erlang, which is called from the `proc_lib_mod` module.

- `edbc_erlc`. This script is equivalent to the `erlc` command used to compile Erlang code from the terminal. The script replaces the Erlang/OTP default parser module with the `edbc_parse_transform` module implemented in EDBC, which parses the contracts ad hoc when translating the source code for execution.

- `edbc_erlcp`. This script compiles Erlang code from the terminal as the `edbc_erlc` script does. Contrarily to `edbc_erlc`, the compilation of the code with this script allow the appearance of contracts, but disables contract checking.

- `edbc_erl`. This script launches an Erlang shell including the set of modules contained in a given directory. Additionally, if indicated, the shell can execute a particular given method after launching, e.g., a set of test cases referring to a contracted function.

- `edbc_edoc`. This script generates the documentation of all the functions of a particular module. The documentation includes, for each contracted

[8]To incorporate improvements introduced in the `gen` module of the Erlang/OTP library, all `gen_mod`, `proc_lib_mod`, and `error_logger_mod` modules must be replaced by their homologous (without the `_mod` suffix) modules in the Erlang/OTP library and adapted to work with EDBC by adding the `_mod` to the corresponding method calls and modifying the returned value of some functions.

function, the information the defined contracts, providing information about the input, output, or functionality expected for the function.

9.2.2 Instrumentation

Note that the code produced by the instrumentation process is standard Erlang code, which can be executed by any standard Erlang runtime system in a completely normal fashion. Technically, the instrumentation is performed using so called Erlang "parse transforms", which permits defining syntactic transformations on Erlang code.

Consider a module with a number of annotated functions. The instrumentation process replaces such annotated functions with a copy of the (possibly modified) original function, together with a number of helper functions that are synthesised from the contracts. The instrumentation is performed in three steps:

1. First, if a function has an associated contract, then an instrumentation to store the relevant information regarding function calls (function name, arguments and stack trace) is performed. This creates a new function which becomes the function entry point, and the original function is renamed. When the new function is called, it stores the call information, and proceeds to call the original function.

2. Then, contracts of type `?DECREASES/1` (including `?SDECREASES/1`) are processed. This instrumentation creates a function which checks if the size of its parameters have decreased between recursive calls. If they are decreased, delayed calls are executed, and if they are not, a contract violation exception is raised. During the instrumentation the original function is also modified by replacing all the recursive calls to calls to the new created function. Note that, due to this instrumentation and the previous one, we have changed the call cycle of a recursive call:

$$\mathtt{f}_{\mathtt{ori}} \rightarrow \mathtt{f}_{\mathtt{ori}} \Rightarrow \mathtt{f}_{\mathtt{si}} \rightarrow \mathtt{f}_{\mathtt{ori}} \rightarrow \mathtt{f}_{\mathtt{dc}} \rightarrow \mathtt{f}_{\mathtt{si}}$$

   where $\mathtt{f}_{\mathtt{ori}}$ is the original function, $\mathtt{f}_{\mathtt{si}}$ is the function that stores the call information, and $\mathtt{f}_{\mathtt{dc}}$ is the function that checks the decreasing of arguments.

3. The remaining contract types are processed distinguishing between contracts of type `?PRE`, and contracts of type `?POST`. All contracts except `?DECREASE` can in fact be generalised to one of of these two types of contract. Of course, each contract has its particularities, however, these particularities do not have any effect in the instrumentation process. The chain of calls becomes

$$\mathtt{f}_{\mathtt{si}} \rightarrow \mathtt{f}_{\mathtt{pre/post}}* \rightarrow \mathtt{f}_{\mathtt{ori}}$$

   where $\mathtt{f}_{\mathtt{pre/post}}*$ are a number of functions (maybe none) introduced by `?PRE`/`?POST` contracts. In the case of a recursive function which defines a

?DECREASE contract, the call chain would be

$$\mathtt{f}_{\mathtt{si}} \rightarrow \mathtt{f}_{\mathtt{pre/post}}* \rightarrow \mathtt{f}_{\mathtt{ori}} \rightarrow \mathtt{f}_{\mathtt{dc}} \rightarrow \mathtt{f}_{\mathtt{si}}$$

Further note that most of the helper functions have a call as its last expression enabling, in this way, so called last call optimisations to reduce the runtime cost of instrumenting code. The only exception is the functions generated for postconditions, which needs to be stacked until internal calls are completely evaluated.

In the rest of the section we explain the particularities of each contract type. Contracts of type ?DECREASE/1, ?PRE/1 and ?POST/1 have not any particularity, i.e. they work as explained in the instrumentation process. The rest of contract types are implemented as described in the following.

- **Execution-time contracts**. These contracts are instrumented as ?PRE/1 contracts because the result, i.e. the parameter of the condition-checker functions of ?POST/1 contracts, is not needed. In fact, the evaluation of the condition-checker function does not return a boolean, but a number which represents the expected time. Then, the delayed call is run in the same process (?EXPECTED_TIME/1) or in a separate one (?TIMEOUT/1). Finally, according to the time needed to run the call, either the result is returned or a contract-violation error is raised (stopping also the evaluation in the case of ?TIMEOUT/1).

- **Purity contracts**. They are also implemented as ?PRE/1 contracts for the same reason as execution-time contracts. In this case, there is not any condition-checker function, so a dummy one is used. In order to check these contracts, we trace all the calls performed during the evaluation of the delayed call as well as receive/send events. The tracing process is performed using the BIF erlang:trace/3. A function call is considered impure if during its evaluation is performed a send, a receive or a call to an impure BIF (checked using the function erl_bifs:is_pure/3).

- **Invariant contracts**. These types of contracts are internally translated to ?POST/1 contracts and attached to functions which can change the state of an Erlang behaviour, e.g. code_change/3, handle_call/3, init/1, etc. Instead of the function result, the behaviour state is used to check whether the synthetized postcondition holds.

- **Type contracts**. For checking the validity of the values according to the expected types we use the Sheriff [82] type checker. Sheriff is run by calling the function sheriff:check/2 which checks whether a given value belongs to a given type. We have gone a step forward making the type checking completely transparent for users and reusing their already-defined type contracts, i.e. their specs. A spec implicitly defines a condition for the parameters and a condition for the result. Therefore, a spec is translated to both a ?PRE/1 and a ?POST/1 contract which internally call to Sheriff and decide form its result whether to continue the evaluation or to raise a contract-violation error with attached details about the violator value and its expected type.

Finally, contract checking can be easily disabled or enabled using a special compilation flag, thus e.g. permitting production code to be compiled and run without instrumentation.

9.3 Related Work

Our contracts are similar to the ones defined in [10], where the function specifications are written in the same language, i.e., Curry, so they are executable. Being executable enables their usage as prototypes when the real implementation is not provided. Their contracts are functions in the source code instead of macros, so it is not clear whether they could be removed in the final release. One of the authors extended this work in [77], where static analysis performed by an SMT solver at compile time was used to check the contracts. This analysis discharged the overhead produced by the dynamic verification of these contracts. In these works, there is not any mention about whether their contracts are integrated with a documentation tool like our contracts are with EDoc. Moreover, they only allow to define basic precondition and postcondition contracts, while we are providing alternative ones like purity or time contracts. Finally, Curry is a pure functional logic language not targeting concurrent computations. For that reason, contracts for purity checking or concurrency behaviours would not be very useful in a language like Curry.

The work in [155] presents a static analysis which infers whether a function is pure or not. Since the focus on the article is on static analysis whereas ours is on dynamic analysis, the purity checking is performed in completely different ways in each work. However, we can benefit from their results by, for instance, avoiding to execute functions that are already known to be impure, reporting earlier to the user a purity-contract violation. In the same way, our system can be used in their approach to check the validity of statically-inferred results.

The type contract language for Erlang [94] allows to specify the intended behaviour of functions. Their usage is twofold: i) as a documentation in the source code which is also used to generate EDoc, and ii) to refine static analyses provided by tools such as Dialyzer. The contract language allows for singleton types, unions of existing types and the definition of new types. However, these types and function specifications do not guarantee type safety. This guarantee comes with Erlang which incorporates strong runtime typing with type errors detected and reported at runtime. Although such a static analysis is quite capable in detecting typing violations, strong typing usually detects unexpected behaviour too far from its source. Therefore, when debugging a program, providing the feature to detect violations of such type contracts at runtime can be a useful aid to provide more precise error location.

The contracts proposed for the concurrent environments follow the same philosophy as the specifications defined in [59, 78]. Indeed, our function `cpre/3` takes its name from these works. Although these works were more focused on enabling the use of formal methods, or testing techniques, to verify nontrivial properties of realistic systems, in this work we demonstrate that they can be used to concisely program server applications which are forced to deal with asynchronous requests that must be delayed, and moreover are also useful for runtime verification.

Dafny [114] is a verification-aware programming language that allows us to define invariants and contracts in their programs. The main difference between their approach and ours is that their contracts are not checked during runtime, but statically. Once you define a contract in Dafny, the system checks whether it is fulfilled in every program compilation as your program grows. This continuous verification is very powerful and consider all the scenarios explicitly in the code, but comes with a higher computational effort. Contrarily, EDBC provides the same type of contracts for Erlang, but checked during runtime. Unfortunately, being runtime checked is double-edged: the compilation is a verification-free process that consumes less resources, but an incorrect use of functions is not noticed until the program is run and the contract is violated during the testing phase. Additionally, as Dafny implements its own programming language, the programmer needs to define complex contracts in a different language that the one used in her implementation. Contrarily, in EDBC, contracts are defined using resources of the native Erlang language, which makes its integration easier for programmers.

The aspect-oriented approach for Erlang (`eAOP`) presented in [29] shares some similarities with our work. eAOP allows the instrumentation of a program with a user-defined tracing protocol (at an expression level). This is able to report events to a monitor (asynchronous) as well as to force some part of the code to block waiting for information from the monitor (synchronicity). Our system could be used to a similar purpose but only at the function level. Additionally, thanks to the functionality freedom allowed in our contracts, EDBC enables the definition of synchronisation operations at the user-defined contracts. More complex modifications of our system, such as the ones done in [126], can transform our work into a complete aspect-oriented system.

Also in Erlang, the work [43] defines a runtime monitoring tool which helps to detect messages which do not match a given specification. These specifications are defined through an automaton, which requires an extra knowledge from the user concerning both the syntax of the specification, and in the whole system operation. We propose a clear and easy way to define the conditions for when to accept a request without needing any user input.

Finally, JErlang [156] enables so called joins in Erlang. This is achieved by modifying the syntax of Erlang receive patterns to permit expressing matching of multiple subsequent messages. Although our goal and theirs are different, both approaches can simplify the way programmers solve similar kinds of problems. Indeed, we could simulate joins by adding a forth parameter to the function `cpre/3`. This additional parameter would represent the still unserved pending requests. When the last request of a user-defined sequence (*join*) is received, the pending requests should be examined to check whether the required join can be served. A similar modification is needed to the callback `handle_call/3` interface so that the pending requests could be served using the `gen_server:reply/2` call.

Part IV

Developed Tools

Chapter 10

Developed Tools and User Guides

In this chapter we show where to find, and how to install and use the tools presented along this thesis. These tools include two program slicers, a regression testing tool, and a library for doing DBC verification in Erlang. The description of each tool is divided in four parts: (i) how to find download and install the tool, (ii) a description about the options and commands to configure and run the tool, and (iii) use cases of configuration and execution of the tool with the description of the result/report produced by the tool.

10.1 JavaSlicer

JavaSlicer is a program slicer for Java implemented in Java that uses the JSysDG as the starting graph. JavaSlicer implements the representation described in Chapter 3 including in the JSysDG the newly defined object-flow and object-reference dependences. The tool gives the user the possibility of selecting any object variable of the program as slicing criterion, obtaining an slice that includes all the necessary statements to compute the value of the object variable itself, including the object it points to (its reference) and the value of all its data members. All the source code of JavaSlicer is publicly available at:

https://github.com/mistupv/JavaSlicer

and a resource-limited web version of the tool can be found at:

https://mist.dsic.upv.es/JavaSlicer/demo

10.1.1 Installation and first steps

To install JavaSlicer in your computer, it is necessary to download the JavaSlicer project from git:

```
~$ git clone https://github.com/mistupv/JavaSlicer
```

Generate Docker testing environment

To build a docker environment for easily testing the tool, enter the folder where the project has been downloaded (ensures that there is a file called Dockerfile inside it) and run these two commands:

```
~$ docker build -t javasdg .
~$ docker run --name javasdg_container -it javasdg bash
```

We will now find ourselves into a new shell where the tool and all its dependences have already been installed. Then, the slicer can be run (see First steps subsection) using the example programs contained in the *examples* folder or creating new java programs via vim text editor.

Manual Installation

JavaSlicer manages its dependencies through maven, so you need to have the JDK (≥ 11) and Maven installed, then run:

```
~$ mvn package -Dmaven.test.skip
```

A fat jar containing all the project's dependencies can be then located at:

```
./sdg-cli/target/sdg-cli-{version}-jar-with-dependencies.jar
```

From now on, for the sake of simplicity, we assume that this jar file has been renamed to `JavaSlicer.jar`.

First steps

JavaSlicer is a program slicer executed from the command line. In order to run the slicer, the user needs to specify a slicing criterion. JavaSlicer is run from the .jar file previously generated with the following command:

```
~$ java -jar JavaSlicer.jar -c file#line:var
```

In the JavaSlicer command, there are two different methods to specify the slicing criterion. The simplest method is with the flag `-c file#line:var`, where `file`, `line`, and `var` must be specified by the user. If the variable appears multiple times in the given line, all of them will be selected.

The tool can be further configured by using different flags. Executing the jar file with the option `-h` (or `--help`) shows the flags accepted by JavaSlicer, which are described hereunder:

- `-f,--file <CriterionFile.java>`. The file that contains the slicing criterion.
- `-l,--line <line-number>`. The line that contains the statement of the slicing criterion.

- `-v,--var <variable-name>`. The name of the variable of the slicing criterion. Not setting this option is equivalent to selecting no variable in the given line number. In case no variable is selected, the slicer will return the slice considering a reachability analysis to the selected statement.
- `-c,--criterion <file#line[:var]>`. The slicing criterion, in the format "file#line:var". The variable is optional. This option may be replaced by `-f`, `-l`, and `-v`. If this argument is set, it overrides the individual ones.
- `-h,--help`. Shows the text containing the information about the command and all the options it accepts.
- `-i,--include <directory[,directory,...]>`. This command is used to include the implementation of classes that are located in other files or directories different from where the slicing criterion is. It includes all the .java files in the directories listed here as part of the dependence graph, letting the analysis and generation of summary edges for all the methods contained in these files. Methods that are not included here or part of the JRE, including third party libraries will not be analysed, resulting in less precise slices.
- `-o,--output <output-dir>`. The directory where the sliced source code should be placed. By default, it is placed at the path `./slice`.
- `-cp,--classpath <jarFile[:jarFile:...]>`. The jar files of all the libraries that need to be included to compile the code to be sliced.

Example 10.1 (Running JavaSlicer)**.** *Consider the Java program shown in Figure 10.1, stored in a file called* `Example1.java` *located in a folder called* `./examples` *in the base directory of the cloned repository. The slice of this*

```
 1  public class Example1 {
 2      public static void main(String[] args) {
 3          int sum = 0;
 4          int prod = 0;
 5          int i;
 6          int n = 10;
 7          for (i = 0; i < 10; i++) {
 8              sum += 1;
 9              prod += n;
10          }
11          System.out.println(sum);
12          System.out.println(prod);
13      }
14  }
```

FIGURE 10.1: Java program to be sliced

program with respect to variable `sum` *in line 11, would be obtained by following command:*

```
~$ java -jar JavaSlicer.jar -c ./examples/Example1.java#11:sum
```

Since no `-o` option has been specified, the slice is stored in the folder `./slice`. This folder contains a file with the same name (`Example1.java`) that contains the corresponding slice, denoted with black code in Figure 10.2.

```
 1  public class Example1 {
 2      public static void main(String[] args) {
 3          int sum = 0;
 4          int prod = 0;
 5          int i;
 6          int n = 10;
 7          for (i = 0; i < 10; i++) {
 8              sum += 1;
 9              prod += n;
10          }
11          System.out.println(sum);
12          System.out.println(prod);
13      }
14  }
```

FIGURE 10.2: Slice for the slicing criterion $\langle 11, sum \rangle$ of the Java code in Figure 10.1

10.1.2 Use case

This section shows an additional example of how JavaSlicer is used via command line with a set of files distributed in different directories. Consider the Java code in Figure 10.3, which contains three different Java classes that make use of some hierarchy properties and are part of the regression test suite used to verify the slicer performance each time a change in introduced[1]. The program creates two different objects of different types and makes a sequence of method calls using these objects as scope. Consider the slicing criterion $\langle 8, b1 \rangle$, where we are interested in the object variable `b1` after the call to method `getA()` in line 8. In order to slice this program, we run the following command:

```
~$ java -jar JavaSlicer.jar -i ./examples/Example2/ -c
./examples/Example2/Example2.java#8:b1 -o ./examples/Example2Output
```

The result of this command is the creation of a new folder in the path specified by the `-o` flag (`./examples/Example2Output`), where the output of JavaSlicer is saved. The output results in a set of files with the same name than the original ones and with the corresponding program slice, shown in Figure 10.4.

In Figure 10.4, all the code related to variable `a1` is completely removed, together with the methods in class `A` that are not called from class `B`. The constructor of class `A` remains in the slices because it is called from the constructor of class `B` with the corresponding `super(...)` call. The same occurs to method

[1]This use case can be found both in `./examples/Example2` or in the regression test suite at: `https://github.com/mistupv/JavaSlicer/tree/develop/sdg-core/src/test/res/regression`

```
 1  public class Example2 {
 2    public static void main(String[] args){
 3      A a1 = new A(1);
 4      B b1 = new B(5.6);
 5      a1.printA();
 6      b1.printA();
 7      b1.updateA(5);
 8      int z = b1.getA();
 9      System.out.print(z);
10    }
11  }
12  public class A {
13    int a = 0;
14    public A(int val) { a = val; }
15    public int getA() { return a; }
16    public void setA(int val) { a = val; }
17    public void printA() {
18      System.out.print(a);
19    }
20    public void updateA(int v) { a++; }
21  }

22  public class B extends A {
23    int b = 5;
24    public B(double val){
25      super((int) val);
26    }
27    public void updateB(B b){
28      b.setB(10);
29    }
30    public int getB() { return b; }
31    public void setB(int val) { b = val; }
32    public void printA() {
33      System.out.print("Useless");
34    }
35    public void updateA(int v) {
36      super.updateA(v);
37      a += v;
38    }
39  }
```

FIGURE 10.3: Three classes used in the use case Example2

```
 1  public class Example2 {
 2    public static void main(String[] args){
 3      A a1 = new A(1);
 4      B b1 = new B(5.6);
 5      a1.printA();
 6      b1.printA();
 7      b1.updateA(5);
 8      int z = b1.getA();
 9      System.out.print(z);
10    }
11  }
12  public class A {
13    int a = 0;
14    public A(int val) { a = val; }
15    public int getA() { return a; }
16    public void setA(int val) { a = val; }
17    public void printA() {
18      System.out.print(a);
19    }
20    public void updateA(int v) { a++; }
21  }

22  public class B extends A {
23    int b = 5;
24    public B(double val){
25      super((int) val);
26    }
27    public void updateB(B b) {
28      b.setB(10);
29    }
30    public int getB() { return b; }
31    public void setB(int val) { b = val; }
32    public void printA() {
33      System.out.print("Useless");
34    }
35    public void updateA(int v) {
36      super.updateA(v);
37      a += v;
38    }
39  }
```

FIGURE 10.4: Slice w.r.t. $\langle 8, b1 \rangle$ of the code in Figure 10.3

updateA() in class A, which is called by its homologous in class B). The execution of method updateA() in class A has a side effect over data member a and, for this reason, needs to be included in the slice. Additionally, method getA() must be also included since we are interested in b1 after the method call and so it needs to be executed. Finally, it is worth to mention that both a and b data members are part of the slice thanks to the corresponding object-flow dependences included by the underlying model.

10.2 e-Knife (a CE-EDG Slicer for Erlang)

e-Knife is a program slicer that operates over the sequential part of the Erlang language. e-Knife is implemented in Java and operates with the EDG as base representation graph. Additionally, the slicer allows for field-sensitive slicing by implementing the constrained-edges approach described in Chapter 4, which supports an accurate management of composite data structures like lists or tuples in Erlang programs. Given an Erlang program and a slicing criterion, e-Knife generates the corresponding CE-EDG and slice it with a high level of precision. All the source code of e-Knife is publicly available at:

https://mist.dsic.upv.es/git/program-slicing/e-knife-erlang

and a resource-limited web version of the tool can be found at:

https://mist.dsic.upv.es/e-knife-constrained/

10.2.1 Installation and first steps

To install e-Knife in your computer, it is necessary to download the e-Knife project from git.

```
~$ git clone https://kaz.dsic.upv.es/git/program-slicing/e-knife-erlang.git
```

Generate Docker testing environment

If you want to generate a docker environment to test the tool, enter the folder where the project has been cloned (ensures that there is a file called Dockerfile inside it) and run these two commands:

```
~$ docker build -t eknife .
~$ docker run --name eKnife_container -it eknife bash
```

After running these two commands, the terminal will enter the docker shell, where the tool and all its dependences have already been installed. The user can now run the tool (see *First steps* subsection) with the benchmarks contained in the "*examples*" folder, or write and save her own programs via vim text editor.

Manual installation

There are some program dependences that need to be considered to install e-Knife in your computer. e-Knife manages its dependences using maven so you need to have the JDK ($\geq$ 11) and Maven installed in your computer. Additionally, during its execution, the tool make successive calls to Erlang modules, so you need to have an Erlang/OTP distribution installed in your computer to execute e-Knife[2]. Finally, a makefile is used to build the project, thus you need to have Make also installed in your computer[3]. When all these dependences are fulfilled, you can now build e-Knife. The whole installation process can be done with the following sequence of commands:

[2]The tool has been developed and tested with the Erlang/OTP 24 version.

[3]The user that has problems installing the tool should visit the *Troubleshooting* section in https://kaz.dsic.upv.es/git/program-slicing/e-knife-erlang for further installation details. This section provides solutions to several communication problems generated by the jinterface Erlang library needed to run e-Knife.

```
~$ cd e-knife-erlang
~$ make release
```

The `make release` command generates a zip file called `e-knife-{version}.zip` with the executable version of the slicer. Additionally, an unzipped folder (called `dist`) is generated with a version of the slicer ready to be run.

First steps

The release version of e-Knife contains a jar file and a folder (ebin) with additional resources used to run the slicer. It is important to keep these resources in the same directory. e-Knife does not provide a GUI, it is run from the command line instead. The used can run the jar file with the `--help` option (or without any argument) to obtain a list and description of the accepted slicer options. The list of options usable when running e-Knife is the following:

- `-i,--input <file/folder>`. The file/folder where the source code is located.
- `-f,--file <file>`. The file (relative to -i) that contains the slicing criterion.
- `-l,--line <num>`. The line of the slicing criterion inside its file.
- `-v,--var <name>`. The variable being sliced inside the given line.
- `-n,--occurrence <num>`. The occurrence of the slicing criterion in the specified line.
- `-o,--output <file/folder>`. The file/folder where the slice will be stored.
- `-G,--print-graph <file.dot>`. Generates a dot file that contains the generated and sliced EDG.
- `-G,--print-graph <file.pdf>`. Generates a pdf file that contains the generated and sliced EDG.
- `--help`. Shows the text containing the information about the options accepted by the command.

Example 10.2. *Consider the Erlang program shown in Figure 10.5, stored in a file called* `example.erl`*. This program is a translation to Erlang of the Java program shown in Figure 10.1.*

The e-Knife slice of this program with respect to variable `Sum` *in line 6, would be obtained with the following command:*

```
~$ java -jar e-knife.jar  -i example.erl -f example.erl
                          -l 6 -v Sum -o slice.txt
```

```
 1  -module(example).
 2  -export([main/0]).
 3
 4  main() ->
 5      {Sum,Prod} = sum_prod(10,10,0,0),
 6      io:format("~p",[Sum]),
 7      io:format("~p",[Prod]).
 8
 9  sum_prod(N,0,S,P) -> {S,P};
10  sum_prod(N,L,S,P) ->
11      S1 = S + 1,
12      P1 = P + N,
13      sum_prod(N,L-1,S1,P1).
```

FIGURE 10.5: Erlang program to be sliced

This command generates the slice with respect to $\langle 6, Sum \rangle$ *in the* ***slice.txt*** *file. It is worth to mention that the slices computed over the CE-EDG may remove from the slice only some elements (subexpressions) of another expression in the code. For this reason, to maintain the computed slice structurally correct (e-Knife generates executable slices), these removed elements are replaced with the Erlang atom "*`sliced`*" in case of expressions and with* `_` *in case of patterns. These replacements are shown in the code with green code. For instance, in the slice of method* ***sum_prod*** *some parameters are not needed (replaced with* `_`*) to compute the value of the slicing criterion, and the corresponding arguments in the* ***sum_prod*** *call are replaced by the atom* ***slice*** *just to keep the method signature.*

```
 1  -module(example).
 2  -export([main/0]).
 3
 4  main() ->
 5      {Sum,_} = sum_prod(sliced,10,0,sliced),
 6      io:format(" p",[Sum]),
 7      io:format(" p",[Prod]).
 8
 9  sum_prod(_, 0, S,_) -> {S,sliced};
10  sum_prod(_, L, S,_) ->
11      S1 = S + 1,
12      P1 = P + N,
13      sum_prod(sliced,L - 1,S1,sliced).
```

FIGURE 10.6: Slice for the slicing criterion $\langle 6, Sum \rangle$ of the Erlang code in Figure 10.5

10.2.2 Use case

This section shows an additional example of how e-Knife is used via command line with a set of files distributed in different directories. Consider the Erlang code in Figure 10.7, which is part of Bencher (Chapter 6, Section 6.4.4), our interprocedural benchmark suite for program slicing.

```
 1  -module(bench14).
 2  -export([main/2]).
 3
 4  main(X,Y) ->
 5    Z = case X of
 6      terminate -> "the end";
 7      {A,B} -> {[A + B,B - A],3};
 8      {3,C} -> g(C);
 9      _ -> {20 * 3,8}
10    end,
11    T = 2,
12    V = f(T) + h(2) + h(3),
13    W = g([X,Y,{X,Y}]),
14    Tuple = {Z,W,V},
15    Tuple.
16
17  g(X) ->
18    [_,_,{R,S}] = X,
19    case R of
20      [1,3] -> 21;
21      [A,B] -> (A * B) / 9;
22      T -> T;
23      _ -> f(4)
24    end.
25  f(7) ->
26    L = 2 + 9,
27    F = L * 3,
28    F + L;
29  f(4) -> 9;
30  f(2) -> 7;
31  f(X) -> X.
32
33  h(X) ->
34    case X of
35      2 -> j({2,4});
36      3 -> k([4,8]);
37      1 -> l(107)
38    end.
39
40  j(A) ->
41    {X,_} = A,
42    X.
43
44  k(B) ->
45    [H|T] = B,
46    H.
47
48  l(C) -> C - 1.
```

FIGURE 10.7: Program called `bench14.erl` of the interprocedural Bencher suite for Erlang

The module contains 7 different methods called from different program points and used to compute the value of three different variables: `V`, `W`, and `Z`. Each one of these variables is computed by using a different set of these methods according to a couple of input parameters received by the `main` method. Consider the variable `Z` in line 14 as the slicing criterion ($\langle 14, Z\rangle$). In order to compute this slice, we run e-Knife with the following command:

```
~$ java -jar e-knife.jar  -i bench14.erl -f bench14.erl
                          -l 14 -v Z -o bench14Slice.erl
```

The result of this command is the creation of a new file in the directory specified in the `-o` flag (`./` in this case), where the output file generated by running e-Knife is saved with the selected name (`bench14Slice.erl`). The output results in a file with the corresponding program slice, shown in Figure 10.8.

As it can be seen in Figure 10.8, the computation of both variables `V` and `W` is never included in the slice, since `Z` is not dependent on them. This fact includes the removal of methods `h`, `j`, `k`, and `l` which are only used in the computation of variable `V`. Additionally, as it can be seen, e-Knife also excludes from the slice method `f` because, despite being called in line 23 inside method `g`, the analysis performed when creating the CE-EDG detects that this case clause is unreachable: the previous pattern `T` corresponds to an unbound variable and would match any possible value of the case expression `R`. Finally, besides removing from the slice all the code in grey, there are two expressions that are also replaced by the `_` symbol to make the slice executable: variable `Y` at line 4

```
1  -module(bench14).
2  -export([main/2]).
3
4  main(X,_) ->
5    Z = case X of
6      terminate -> "the end";
7      {A,B} -> {[A + B,B - A],3};
8      {3,C} -> g(C);
9      _ -> {20 * 3,8}
10   end,
11   T = 2,
12   V = f(T) + h(2) + h(3),
13   W = g([X,Y,{X,Y}]),
14   Tuple = {Z,W,V},
15   Tuple.
16
17 g(X) ->
18   [_,_,{R,_}] = X,
19   case R of
20     [1,3] -> 21;
21     [A,B] -> (A * B) / 9;
22     T -> T;
23     _ -> f(4)
24   end.
25 f(7) ->
26   L = 2 + 9,
27   F = L * 3,
28   F + L;
29 f(4) -> 9;
30 f(2) -> 7;
31 f(X) -> X.
32
33 h(X) ->
34   case X of
35     2 -> j({2,4});
36     3 -> k([4,8]);
37     1 -> l(107)
38   end.
39
40 j(A) ->
41   {X,_} = A,
42   X.
43
44 k(B) ->
45   [H|T] = B,
46   H.
47
48 l(C) -> C - 1.
```

FIGURE 10.8: Slice w.r.t. ⟨14, Z⟩ obtained after running e-Knife for the code in Figure 10.7

and variable S at line 18, which are not used to compute the value of the slicing criterion Z.

10.3 SecEr

SecEr (Software Evolution Control for Erlang) is an implementation of the POI Testing approach described in Chapter 8. SecEr is a tool for Erlang able to automatically generate a test suite to check the behaviour of a point of interest. It can be used for regression testing in two different ways: (i) generating a test suite for a future comparison or (ii) by automatically comparing two releases of an Erlang program. The comparison performed by SecEr focuses on a set of program points specified by the user. The tool automatically generates a test suite that checks the behaviour of those program points. These test cases try to maximise the branch coverage, covering a large quantity of different executions, and the tool notifies the user about any unexpected result. SecEr is open source and is publicly available at:

https://github.com/mistupv/secer

10.3.1 Installation and first steps

SecEr makes use of the Erlang tools `TypEr`, `PropEr`, and `CutEr`. SecEr includes a version of these tools to ease its installation process. In the case of this tool, the way of cloning the repository differs if we generate the docker environment or perform a manual installation.

Generate Docker testing environment

In this case, the docker environment can be generated by the following four commands:

```
~$ git clone https://github.com/mistupv/secer.git
~$ cd secer
~$ docker build -t secer .
~$ docker run --name secer_container -it secer bash
```

After running the docker image in the container, a new shell environment is entered, where the `secer` command (explained in the following sections) can be used from any directory.

Manual installation

To manually install SecEr, all `CutEr` external dependencies need to be also fulfilled (`https://github.com/cuter-testing/cuter`). Additionally, the repository must be compiled with the `--recursive` option. The source code of the SecEr tool, together with a set of benchmarks and examples to start using the tool can be found and downloaded from the *mistupv* github repository by following these commands:

```
~$ git clone --recursive https://github.com/mistupv/secer.git
~$ cd secer
~$ make
~$ export PATH=$PATH:/path/to/secer/
```

First Steps

In this section, we describe SecEr and how to use it to automatically compare two program versions or obtain test cases from a source code. First, we show an example of a configuration file describing the different parts it contains. SecEr provides an API that internally implements different configuration modes, leaving to the user the definition of comparison functions. Then, we illustrate how SecEr is called from the command line, and finally we show some use cases that illustrate how SecEr is used.

Configuration file for POI testing

To explain the SecEr configuration process easier, we use a particular example of two programs versions that solve the *happy numbers* problem (Figure 10.9). A *happy number* can be defined as a number which will yield 1 when it is replaced by the sum of the square of its digits repeatedly. If this process results in an endless cycle of numbers containing 4, then the number is called an *unhappy number*. For instance, number 10 would be a happy number ($10 \rightarrow 1^2 + 0^2 = 1$) while number 11 would be an unhappy number ($11 \rightarrow 1^2 + 1^2 = 2 \rightarrow 2^2 = 4$).

```erlang
-module(happy1).
-export[main/2].

-spec main(pos_integer(),pos_integer()) ->
  [pos_integer()].
main(N, M) ->
  happy_list(N, M, []).

happy_list(_, N, L) when length(L) =:= N ->
  lists:reverse(L);
happy_list(X, N, L) ->
  Happy = is_happy(X),
  if Happy ->
    happy_list(X + 1, N, [X|L]);
  true ->
    happy_list(X + 1, N, L)
  end.

is_happy(1) -> true;
is_happy(4) -> false;
is_happy(N) when N > 0 ->
  N_As_Digits =
    [Y - 48 ||
    Y <- integer_to_list(N)],
  is_happy(
    lists:foldl(
      fun(X, Sum) ->
        (X * X) + Sum
      end,
      0,
      N_As_Digits));
is_happy(_) -> false.
```

(A) happy1.erl

```erlang
-module(happy2).
-export[main/2].

is_happy(X, XS) ->
  if
    X == 1 -> true;
    X < 1 -> false;
    true ->
      case member(X, XS) of
        true -> false;
        false ->
          is_happy(sum(map(fun(Z) -> Z*Z end,
            [Y - 48
              || Y <- integer_to_list(X)])),
            [X|XS])
      end
  end.

happy(X, Top, XS) ->
  if
    length(XS) == Top -> sort(XS);
    true ->
      case is_happy(X,[]) of
        true -> happy(X + 1, Top, [X|XS]);
        false -> happy(X + 1,Top, XS)
      end
  end.

-spec main(pos_integer(),pos_integer()) ->
  [pos_integer()].
main(N, M) ->
  happy(N, M, []).
```

(B) happy2.erl

FIGURE 10.9: Two different versions of a program to compute happy numbers in Erlang

Both program versions solve the problem to find the next M happy numbers starting from a particular positive integer N.

SecEr permits to use configuration files that can be reused in different invocations. A configuration file is an Erlang program formed by a set of functions that can be invoked from the SecEr command. For instance, the configuration module `test_happy` of Figure 10.10 allows the use of functions `relResult/0`, `relIsHappy/0`, `funs/0`, and `config/0` when calling SecEr from the command line. In this figure, we can see that POIs can be specified in two different ways: (i) with a tuple with the format `{'FileName', Line, Expression`[4]`, Occurrence}` as shown in Figure 10.10 line 6, and (ii) with a tuple `{'FileName', {InitialLine, InitialColumn}, {FinalLine, FinalColumn}}`[5] representing the initial and final line and column in the corresponding program file. This notation is shown in line 8.

To ease the definition of comparison functions in the configuration file, SecEr provides an API (module `secer_api`) that implements different utilities

[4]Expressions with an specific name, e.g. variables, are denoted by a tuple `{var,'VarName'}`. Note that expressions denoted by reserved Erlang words (`case`, `if`...) must be specified in single quotation marks.

[5]POIs of this type are internally translated to POIs of the first type.

```
1  -module(test_happy).
2  -export([relResult/0,relIsHappy/0,
3          funs/0,config/0,]).
4
5  poiResult1() ->
6      {'happy1.erl',7,call,1}.
7  poiResult2() ->
8      {'happy2.erl',{32,2},{32,16}}.
9
10 poiIsHappy1() ->
11     {'happy1.erl',12,call,1}.
12 poiIsHappy2() ->
13     {'happy2.erl',23,call,1}.
14
15 relResult() ->
16     [{poiResult1(),poiResult2()}].
17
18 relIsHappy() ->
19     [{poiIsHappy1(),poiIsHappy2()}].
20 funs() ->
21     "[main/2]".
22
23 config() ->
24     secer_api:nuai_tr_config(mytecf(),myubrm()).
25
26 mytecf() ->
27     fun(T1E,T2E) ->
28       if (vef()(T1E)) == vef()(T2E)) -> true;
29           (secer_api:get_ca(T1E) ==
30            secer_api:get_ca(T2E)) ->
31               diff_value_same_args;
32        true -> diff_value_diff_args
33     end.
34
35 myubrm() ->
36   [{diff_value_same_args,[val,ca]},
37    {diff_value_diff_args,[val,ca]}].
38
39 vef() -> secer_api:vef_value_only().
```

FIGURE 10.10: Configuration file to test happy modules

like the comparison modes described in Section 8.2.1. For instance, in Figure 10.10, line 24 defines a configuration for tracing function calls by using function `secer_api:nuai_tr_config/2`, which represents the configuration 1c shown in Section 8.2.1. This configuration requires a comparison function and a customised report for each defined error type. In this case it uses `mytecf/0` to compare trace elements, which uses the value of the arguments at the call (lines 29 and 30) to classify errors. Finally, the error message is customised by `myubrm/0`, which indicates the part of the additional information to be shown when a particular error (the ones defined in `mytecf/0` function) is detected. In `myubrm/0` code in Figure 10.10, `val` stands for the POI value and `ca` for the arguments of the function calls selected as POI.

Calling SecEr

In this section we describe how, given two versions of the same program, SecEr can be used to check behavioural changes in the code. First, we show how to call SecEr from the command line, then, we show how the tool is run with the configuration described in Figure 10.10 to analyse the differences of both implementations of the happy numbers programs given in Figure 10.9. Finally, we show how SecEr can be also used to obtain a set of test cases given just one version of a program.

First of all, we need to show the SecEr command and the arguments it requires. The structure of the command is shown in Figure 10.11.

```
$ secer -pois "PAIRS_OF_POIS"
         [-suite "POI"]
         [-funs "INPUT_FUNS"]
          -to TIMEOUT
         [-config "CONFIG_FUN"]
```

FIGURE 10.11: SecEr command format

SecEr receives four arguments, but those arguments in brackets are optional.

- **`-pois "PAIRS_OF_POIS".`** This argument contains a list with the pairs of POIs that must be compared during the execution. It must be a string (`""`) and can contain the explicit reference to the file, line and occurrence of each POI, or a call to a function containing this information in the configuration file, but must be always expressed by means of Erlang code.

- **`-suite "POI".`** This argument contains a POI defined over a particular program version. The idea of this flag is to store in a separate file all the test cases generated for every input function when trying to evaluate the POI. These test cases are stored in the suite folder inside SecEr main directory, where the user can extract all the input calls generated to maximise the branch coverage of the given Erlang program.

- **`-funs "INPUT_FUNS".`** This argument contains a list of functions in the form `function_name/arity`. Like the previous argument, it must be a string containing the corresponding Erlang code and can also be a call to a function to the configuration file. If no list of functions is given, SecEr selects the set of exported functions in the module as input functions.

- **`-to TIMEOUT.`** This argument is an integer that represents the amount of time (in seconds) that SecEr can use to generate test cases for each function given in the `-fun` argument.

- **`-config "CONFIG_FUN".`** This argument contains the configuration selected to run SecEr. This configuration defines how POIs are compared according to their computed values, to the value of some part of the additional information, or to any kind of customised criteria defined by the user. An example of this configuration is the `config` function defined in Figure 10.10. If no argument is provided, SecEr considers only the equality between the values computed for both POI traces as described in the comparison approach 1 of Section 8.4.2. As the two first arguments, this argument must contain a string with Erlang code.

By default, the traces of the POIs are compared using the standard equality and in the order they appear during the program execution, as it is defined by the first comparison function in Section 8.4.2. Alternatively, we can customise our comparison by isolating the traces of different POIs. This customisation is specially useful to test multiple unrelated POIs in a single run. Among the functions provided in the `secer_api` module, we provide different configuration modes like, for instance, the mode that compares the traces independently (`secer_api:cf_independent`) as it is described in comparison approaches 2 and 3 in Section 8.4.2. The difference between both comparison modes is shown in Example 10.3.

Example 10.3. *Consider two POIs, POI_1 and POI_2 in the original code, and their counterparts POI'_1 and POI'_2 in the new code. If an execution executes the POIs in the following order:*

Original code: $POI_1 = 42 \ldots POI_1 = 43 \ldots POI_1 = 50 \ldots POI_2 = 0$
New code: $POI'_1 = 42 \ldots POI'_1 = 43 \ldots POI'_2 = 0 \ldots POI'_1 = 50$

If we use the ***cf_independent*** *configuration option, SecEr would record the traces*

Trace POI_1 *= [42,43,50]* *Trace* POI'_1 *= [42,43,50]*
Trace POI_2 *= [0]* *Trace* POI'_2 *= [0]*

and will report that there are no discrepancies between the POIs. In contrast, if no configuration is specified, SecEr will consider the execution order of the POIs, and will alert that this order has changed.

Note that, in the implementation, the limit used to stop generating test cases is a timeout, while the formalization of the technique uses a number to specify the amount of test cases that must be generated (see variable *top* in Section 8.3.4). This is not a limitation, but a design decision to increase the usability of the tool. The user cannot know a priory how much time it could take to generate an arbitrary number of test cases. Hence, to make the tool predictable and give the user control over the computation time, we use a timeout. Thus, SecEr generates as many test cases as the specified timeout permits.

Finally, SecEr can also be used to generate and store a suite of tests given a program an a set of POIs. If we want to run the command to only generate a test suite, we need to provide a list of POIs (`LIST_OF_POIS`) contained in double quotes, a list of initial functions (`INPUT_FUNCTIONS`), and a timeout (`TIMEOUT`). An example of SecEr invocation with two POIs is the following:

```
$ secer -suite "test_happy:poiResult1()" -funs "test_happy:funs()"
          -to 10
```

SecEr report

When SecEr is called from the command line to compare two program versions, a report is always generated at the end of its execution. There are two possibilities: both versions of the code present the same behaviour for all the generated test cases, or n test cases failed during the comparison phase of the execution of both versions. Depending on the scenario the message SecEr provides is completely different. When comparison phase goes smoothly, a simple notification message is given to the user, providing the number of executed test cases and the positive result obtained.

```
$ secer -pois "test_happy:relResult()" -funs "test_happy:funs()"
          -to 10 -config "test_happy:config()"
```

FIGURE 10.12: SecEr command to compare the happy number programs of Figure 10.9

Example 10.4. *Consider the comparison of both implementations of the happy numbers problem given in Figure 10.9. Then consider the SecEr command in Figure 10.12 which makes use of the configuration file defined in Figure 10.10. When we run this command, SecEr returns the result in Figure 10.13.*

```
$ secer -pois "test_happy:relResult()" -funs "test_happy:funs()"
          -to 10 -config "test_happy:config()"

Function: main/2
----------------------------
Generated test cases: 1142
Both versions of the program generate the expected result for the
defined POIs
```

FIGURE 10.13: Result of calling SecEr with the configuration file defined in Figure 10.10

The message shows that, with the provided comparison configuration, the values of each pair of POIs have been the same for 1142 different test cases. These 1142 test cases have been generated by SecEr for the input function `main/2` *selected in the* `test_happy` *configuration file during the 10 seconds' timeout given in the* `-to` *argument.*

On the other hand, when any test case fails SecEr provides a report with more information in order to ease the location of the error source. This report can be augmented if any unexpected behaviour report message (function `myubrm` previously seen in some configuration files) is provided by the user for the kind of error given.

Example 10.5. *In order to see the output of the tool when the behaviours of the two compared programs differ, we have introduced an error inside the* `is_happy/2` *function of* `happy2` *module in Figure 10.9b. The error is introduced by replacing the whole line 7:*

```
X < 1 -> false; ⇒  X < 10 -> false;
```

With this change, the behaviour of both programs differs. When the user reruns SecEr using the new program, it reports the error message shown in Figure 10.14.

This message contains much more information than the previous report. In this case, it contains the number of generated test cases but also the amount of them that generate different execution values for the given POIs. These mismatching test cases are classified by the kind of error they generated (in this case all the errors were of the type: `different_value_same_args`*) and an example of the test case that generated this kind of error is given as extra information (call to input function* `main(5,8)` *in the example). Then the given report is further augmented with the computed trace and any extra information extracted to the additional information by the configuration given to this particular type of error. In the example, since both POIs were method calls, the report shows*

```
$ secer -pois "test_happy:relResult()" -funs "test_happy:funs()"
         -to 10 -config "test_happy:config()"

Function: main/2
----------------------------
Generated test cases: 1143
Mismatching test cases: 45 (3.93%)
  Error Types:
    + different_value_same_args => 45 Errors
        Example call: main(5,8)

------ Detected Error ------
Call: main(5,8)
Error Type: different_value_same_args
- - - - - - - - - - - - - - -
POI: {'happy1.erl',7,call,1}
  Trace:
    [[7,10,13,19,23,28,31,32]]
  Call POI Info:
    Callee: happy_list
    Args: [5,8,[]]
POI: {'happy2.erl',32,call,1}
  Trace:
    [[10,13,19,23,28,31,32,44]]
  Call POI Info:
    Callee: happy
    Args: [5,8,[]]
```

FIGURE 10.14: Example of SecEr error message when an unexpected behaviour is found

the trace values and the callee and the list of arguments of the call, which may be handy to detect the source of the error.

10.3.2 Use cases

In this section, we show how POI testing can be used to find the location of a difference introduced during the software development phase. For this purpose, we apply POI testing iteratively. First of all, we select the POIs we want to compare, and we move the POI across the program in different executions to exclude possible error causes. We repeat this process until we find the instruction that generates the unexpected behaviour.

Calling traces

This use case illustrates how the call tracing is used to find out the source of a discrepancy. In particular, we compare two versions of an Erlang program that aligns columns of a string with multiple lines. The code of both versions is shown in Figure 10.15. There is only one difference between both code versions. While `align_columns_ok.erl` version code is implemented with line 31 of Figure 10.15, `align_columns.erl` version replace that line of code with line

32. Both program versions are part of the benchmarks used in EDD (Erlang Declarative Debugger) [26].

```
 1  -module (align_columns_ok). / -module (align_columns).
 2  -export([align_left/0, align_right/0, align_center/0]).
 3
 4  align_left()-> align_columns(left).
 5  align_right()-> align_columns(right).
 6  align_center()-> align_columns(centre).
 7  align_columns(Alignment) ->
 8      Lines =
 9          ["Given$a$text$file$of$many$lines$where$fields$within$a$line$",
10           "are$delineated$by$a$single$'dollar'$character,$write$a$program",
11           "that$aligns$each$column$of$fields"],
12      Words = [ string:tokens(Line, "$") || Line <- Lines ],
13      Words_length = lists:foldl( fun max_length/2, [], Words),
14      [prepare_line(Words_line, Words_length, Alignment)
15       || Words_line <- Words].
16
17  max_length(Words_of_a_line, Acc_maxlength) ->
18      Line_lengths = [length(W) || W <- Words_of_a_line ],
19      Max_nb_of_length = lists:max([length(Acc_maxlength), length(Line_lengths)]),
20      Line_lengths_prepared = adjust_list(Line_lengths, Max_nb_of_length, 0),
21      Acc_maxlength_prepared = adjust_list(Acc_maxlength, Max_nb_of_length, 0),
22      Two_lengths =lists:zip(Line_lengths_prepared, Acc_maxlength_prepared),
23      [ lists:max([A, B]) || {A, B} <- Two_lengths].
24
25  adjust_list(L, Desired_length, Elem) ->
26      L++lists:duplicate(Desired_length - length(L), Elem).
27
28  prepare_line(Words_line, Words_length, Alignment) ->
29      All_words = adjust_list(Words_line, length(Words_length), ""),
30      Zipped = lists:zip(All_words, Words_length),
31      [ apply(string, Alignment, [Word, Length + 1, $\s]) %align_columns_ok
32      [ apply(string, Alignment, [Word, Length - 1, $\s]) %align_columns
33        || {Word, Length} <- Zipped ].
```

FIGURE 10.15: Align columns program versions

They can be found at: https://github.com/tamarit/edd/tree/master/examples/align_columns. These programs only export three functions with zero parameters. Thus, they can be considered unit cases. We could use any or all of these functions as starting point for the behaviour comparison process, but, for simplicity, we will focus on just one of these three functions. The SecEr's configuration file (Figure 10.16), defines that the function selected as input function is `align_left/0` (Figure 10.16, line 10).

In order to test the behaviour preservation between both versions with SecEr, it is a common practice to start by selecting as POI the last expression of each input function. Therefore, we select the POIs defined in line 3, which are paired by the relation defined in function `rel1()` in line 7 of Figure 10.16. The execution of SecEr shown in Figure 10.17 reveals an unexpected behaviour found in the execution of `align_left/0`.The current POI is a static call, for this reason we do not define any specific configuration for this run, we only compare the result of both method calls. In this case, the misbehaviour given can only be generated by the implementation of the function called. Then, we should look for the error source inside the function `align_columns/1` (line 7, Figure 10.15).

Therefore, in order to find the error source, we move the POI to the last expression of function `align_columns/1`, concretely to the call to `prepare_line/3`

```
 1 -module (test_align).
 2 -compile(export_all).
 3 poi1Old() -> {‘align_columns_ok.erl’, 4, call}.  poi1New() -> {‘align_columns.erl’, 4, call}.
 4 poi2Old() -> {‘align_columns_ok.erl’, 14, call}. poi2New() -> {‘align_columns.erl’, 14, call}.
 5 poi3Old() -> {‘align_columns_ok.erl’, 31, call}. poi3New() -> {‘align_columns.erl’, 32, call}.
 6
 7 rel1() -> [{poi1Old(),poi1New()}].
 8 rel2() -> [{poi2Old(),poi2New()}].
 9 rel3() -> [{poi3Old(),poi3New()}].
10 funs() -> "[align_left/0]".
11
12 config() -> secer_api:nuai_tr_config(mytecf(),ubrm()).
13 mytecf() ->
14   fun(TO,TN) -> VEF = secer_api:vef_value_only(),
15                 case VEF(TO) == VEF(TN) of
16                      true -> true;
17                    false ->
18                      case secer_api:get_te_args(TO) == secer_api:get_te_args(TN) of
19                        true -> different_value_same_args;
20                        false -> different_value_different_args
21                      end
22                 end
23   end.
24 ubrm() -> [{different_value_same_args,[val,ca]},{different_value_different_args,[val,ca]}].
```

FIGURE 10.16: Align columns configuration file

inside the list comprehension. Now, since this call is not static, we take profit of the additional information provided by using the configuration defined by function `config/0` (line 12 of Figure 10.16). This function uses the enhanced call information to classify and report new types of errors. Another unexpected behaviour is reported by SecEr for this pair of POIs. However, as we can observe in Figure 10.18, this result has been extended with specific call information. The reported error is `different_value_same_args`. Thus, it indicates that both versions performed exactly the same call. Therefore, according to this report, the error source must be inside the function `prepare_line/3` (line 28, Figure 10.15).

Then, we define a new POI inside the `prepare_line` code. Instead of selecting as POI its last expression, i.e. the list comprehension, which we already know that behaves different, we select the expression inside this list comprehension, i.e., the call to function `apply/3` (line 31/32, Figure 10.15). Being this expression a function call, we can reuse the previous configuration. Figure 10.19 shows the report provided by SecEr with this configuration. Note that the reported misbehaviour is `different_value_different_args` now. This means that there is a discrepancy in one of the arguments between the versions. Finally, we can easily find out what argument is the error source by looking at the reported example provided by SecEr.

Stack traces

This use case demonstrates how the stack trace can help us when looking for an unexpected behaviour source. Suppose that we are comparing the behaviour of two different versions of an Erlang program that implement the mergesort algorithm. The code of both versions is shown in Figure 10.20, where there is just one difference between both code versions. While `merge_ok.erl` version code is implemented with line 23 of Figure 10.20, `merge.erl` version replaces this line

```
$ secer -pois "test_align:rel1()" -funs "test_align:funs()" -to 5

Function: align_left/0
----------------------------
Generated test cases: 1
Mismatching test cases: 1 (100.0%)
  Error Types:
    + different_value => 1 Errors
        Example call: align_left()

------ Detected Error ------
Call: align_left()
Error Type: different_value
- - - - - - - - - - - - - - -
POI: {'align_columns_ok.erl',4,call,1}
  Trace:
    [[["Given ","a   ","text ","file ","of  ","many  ",
      "lines  ","where ","fields ","within ","a ","line "],
      ["are ","delineated ","by ","a  ","single ","'dollar' ",
       "character, ","write ","a  ","program "," "," "],
      ["that ","aligns  ","each ","column ","of  ","fields ",
       "   ","  ","  ","  "," ","  "]]]

POI: {'align_columns.erl',4,call,1}
  Trace:
    [[["Give","a  ","tex","file ","of ","many ","lines ","wher",
       "field","within",[],"lin"],
      ["are ","delineate","by ","a ","singl","'dollar","character","writ",
       "a ","progra",[]," "],
      ["that","aligns ","eac","colum","of ","fields ","   "," ",
       "  ","  ",[]," "]]]
----------------------------
```

FIGURE 10.17: SecEr report for the function call in line 14 as POI

of code with line 24. Both program versions are part of the benchmarks used in EDD (Erlang Declarative Debugger) [26]. They can be found at: `https://github.com/tamarit/edd/tree/master/examples/mergesort`. Both programs export the function `mergesortcomp/1`, which has only one parameter, i.e., a list of integers. Therefore, this function is defined as input function in the configuration file (line 7 of Figure 10.21).

In this case, we are not defining a custom TECF as we did in the previous use case. Instead, we are just adding stack trace information to the final report of the detected errors. For this purpose, we take profit of the defined function `secer_api:nuai_r_config/1` (line 9, Figure 10.21), which represents the scenario 1b in Section 8.2.1, including information about the stack trace when reporting the error.

The difference between these two versions is one of the recursive clauses of function `merge/3`. Therefore, it makes sense to select as POI the call to this function in line 17 of the Figure 10.20. Figure 10.22 shows the report given by running SecEr with this configuration. This report indicates that there is an unexpected behaviour. Using the report, we can notice that the forth evaluation

```
$ secer -pois "test_align:rel2()" -funs "test_align:funs()"
          -to 5 -config "test_align:config()"
Function: align_left/0
----------------------------
Generated test cases: 1
Mismatching test cases: 1 (100.0%)
  Error Types:
    + different_value_same_args => 1 Errors
        Example call: align_left()
------ Detected Error ------
Call: align_left()
Error Type: different_value_same_args
- - - - - - - - - - - - - - -
POI: {'align_columns_ok.erl',14,call,1}
  Trace:
    [["Given","a   ","text","file ","of  ","many  ",
      "lines  ","where","fields","within","a","line"]]
  Call POI Info:
    Callee: prepare_line
    Args: [["Given","a","text","file","of","many","lines",
             "where","fields","within","a","line"],
           [5,10,4,6,6,8,10,5,6,7,1,4],left]
POI: {'align_columns.erl',14,call,1}
  Trace:
    [["Give","a   ","tex","file","of ","many ","lines  ","wher",
      "field","within",[],"lin"]]
  Call POI Info:
    Callee: prepare_line
    Args: [["Given","a","text","file","of","many","lines",
             "where","fields","within","a","line"],
           [5,10,4,6,6,8,10,5,6,7,1,4],left]
----------------------------
```

FIGURE 10.18: SecEr reports UB from call to `prepare_line` in line 14 as POI

of the POI differs between both versions, while their stack is the same.

Therefore, in order to know whether the error is in the arguments of the call or in the called function, we run SecEr with a configuration such as the one used in the previous use case. For clarity, we omit the details of this step here. The unexpected behaviour report indicates that the problem is inside the called function.

Then, the next step is to place a POI inside the function `merge/3` (line 18, Figure 10.20). We choose the clause that contains the recursive calls (line 21, Figure 10.20) because it is the most visited clause during the evaluation. In particular, we place the POI in the case expression in line 22 of Figure 10.20. When we rerun SecEr, we obtain the report shown in Figure 10.23. This report provides some interesting information about both POIs. The behaviour discrepancy has been detected in the values computed in the fifth evaluation of the POI. Additionally, both stack traces differ. In the stack trace produced by the old version there are two stacked calls to function `merge/2` while in the stack trace of the new one there is only one. This means that the old version is performing an extra recursive call before reaching the base case. Then, the

```
$ secer -pois "test_align:rel2()" -funs "test_align:funs()"
          -to 5 -config "test_align:config2()"

Function: align_left/0
----------------------------
Generated test cases: 1
Mismatching test cases: 1 (100.0%)
  Error Types:
    + different_value_different_args => 1 Errors
        Example call: align_left()

------ Detected Error ------
Call: align_left()
Error Type: different_value_different_args
- - - - - - - - - - - - - - -
POI: {'align_columns_ok.erl',31,call,1}
  Trace:
    ["Given  "]
  Call POI Info:
    Callee: apply
    Args: [string,left,["Given",11,32]]

POI: {'align_columns.erl',32,call,1}
  Trace:
    ["Given  "]
  Call POI Info:
    Callee: apply
    Args: [string,left,["Given",9,32]]
----------------------------
```

FIGURE 10.19: SecEr reports UB from call to `apply` in lines 31/32 as POI

```
 1  -module (merge_ok)./-module (merge).
 2  -export([mergesortcomp/1]).
 3
 4  -spec mergesortcomp([integer()]) ->
 5    any().
 6  mergesortcomp(List) ->
 7    mergesort(List, fun comp/2).
 8
 9  mergesort([], _Comp) -> [];
10  mergesort([X], _Comp) -> [X];
11  mergesort(L, Comp) ->
12    Half = length(L) div 2,
13    L1 = take(Half, L),
14    L2 = last(length(L) - Half, L),
15    LOrd1 = mergesort(L1, Comp),
16    LOrd2 = mergesort(L2, Comp),
17    merge(LOrd1, LOrd2, Comp).
18  merge([], [], _Comp) -> [];
19  merge([], S2, _Comp) -> S2;
20  merge(S1, [], _Comp) -> S1;
21  merge([H1 | T1], [H2 | T2], Comp) ->
22    case Comp(H1,H2) of
23      false -> [H2 | merge([H1 | T1], T2, Comp)]; % merge_ok
24      false -> [H2 | merge(T1 ++ [H1], T2, Comp)]; % merge
25      true -> [H1 | merge(T1, [H2 | T2], Comp)]
26    end.
27
28  comp(X,Y) -> X < Y.
29
30  take(0,_) -> [];
31  take(1,[H|_])-> [H];
32  take(_,[])-> [];
33  take(N,[H|T])-> [H | take(N-1, T)].
34
35  last(N, List) ->
36    lists:reverse(take(N, lists:reverse(List))).
```

FIGURE 10.20: Mergesort program versions

final step is to place a POI in each recursive call observing the values of their arguments as in the previous case. With the report provided by SecEr when using this configuration, users can easily spot the error source.

```
-module (test_mergesort).
poi1Old() -> {'merge_ok.erl', 17, call, 1}.    poi1New() -> {'merge.erl',17, call, 1}.
poi2Old() -> {'merge_ok.erl', 22, 'case', 1}.  poi2New() -> {'merge.erl', 22, 'case', 1}.

rel1() -> [{poi1Old(),poi1New()}].
rel2() -> [{poi2Old(),poi2New()}].
funs() -> "[mergesortcomp/1]".

config() -> secer_api:nuai_r_config([{different_value,[val,st]}]).
```

FIGURE 10.21: Mergesort configuration file

```
$ secer -pois "test_mergesort:rel1()" -funs "test_mergesort:funs()"
          -to 5 -config "test_mergesort:config()"

Function: mergesortcomp/1
----------------------------
Generated test cases: 5692
Mismatching test cases: 3369 (59.18%)
  Error Types:
    + different_value => 3369 Errors
        Example call: mergesortcomp([0,-1,1,2,-3])

------ Detected Error ------
Call: mergesortcomp([0,-1,1,2,-3])
Error Type: different_value
- - - - - - - - - - - - - - -
POI: {'merge_ok.erl',17,call,1}
  Trace:
    [[-1,0],[-3,2],[-3,1,2],[-3,-1,0,1,2]]
  Stack
    {merge_ok,mergesort,2,{line,17}}

POI: {'merge.erl',17,call,1}
  Trace:
    [[-1,0],[-3,2],[-3,1,2],[-3,0,-1,1,2]]
  Stack
    {merge,mergesort,2,{line,17}}
----------------------------
```

FIGURE 10.22: SecEr reports UB from call to `merge` in line 17 of Figure 10.20 as POI

Concurrent Environments

In this use case, we see how SecEr proves to be useful also in concurrent programs. Consider the code in Figure 10.24, which shows a fragment of the `gen_server` defined in:

https://github.com/hcvst/erlang-otp-tutorial#otp-gen_server

The server's state is simply a counter that tracks the number of requests served so far. The server defines three types of requests through the functions `handle_call` and `handle_cast`:

1. The synchronous request (i.e., a request where the client waits for a reply) `get_count`, which returns the current server's state.

```
$ secer -pois "test_mergesort:rel2()" -funs "test_mergesort:funs()"
          -to 5 -config "test_mergesort:config()"

Function: mergesortcomp/1
----------------------------
Generated test cases: 4878
Mismatching test cases: 2885 (59.14%)
  Error Types:
    + different_value => 2885 Errors
        Example call: mergesortcomp([5,-6,-6,2,3])

------ Detected Error ------
Call: mergesortcomp([5,-6,-6,2,3])
Error Type: different_value
- - - - - - - - - - - - - - -
POI: {'merge_ok.erl',22,'case',1}
  Trace:
    [[-6,5],[2,3],[-6,2,3],[3,5],[2,3,5]]
  Stack
    {merge_ok,merge,3,{line,25}}
    {merge_ok,merge,3,{line,23}}

POI: {'merge.erl',22,'case',1}
  Trace:
    [[-6,5],[2,3],[-6,2,3],[3,5],[-6,3,5]]
  Stack
    {merge,merge,3,{line,24}}
----------------------------
```

FIGURE 10.23: SecEr report when using case expression in line 22 of Figure 10.20 as POI

2. The asynchronous request (i.e., a request where the client does not wait for a reply) `stop`, which stops the server.

3. The asynchronous request `say_hello`, which makes the server print hello in the standard output.

The first and the third requests modify the server's state by adding one to the total number of requests served so far. The second one does not modify the state but rather it returns a special term that makes the `gen_server` stop itself.

To illustrate how SecEr can detect an unexpected behaviour change between two versions of the code, consider that the current (buggy) version is the one depicted in Figure 10.24, while the (correct) original version of the code contains line 34 instead of line 35.

Then, we can define a configuration file like the one in Figure 10.25 and run SecEr to see whether the behaviour is preserved or not. This configuration file uses two input functions (`handle_call` and `handle_cast`), and a POI relation that defines three POIs, one for each request output. If we run SecEr using this configuration we obtain the output showed at Figure 10.26. In the output we can see that no errors are reported for function `handle_call`, which means that the request `get_count` is served in the same way in both versions. In contrast,

```
 1  -module(hello_server).
 2  -behavior(gen_server).
 3  -record(state, {count}).
 4
 5  -export([
 6    init/1,
 7    terminate/2,
 8    handle_call/3,
 9    handle_cast/2]).
10
11  init([]) ->
12    {ok, #state{count=0}}.
13
14  terminate(_Reason, _State) ->
15    error_logger:
16      info_msg("terminating~n"),
17    ok.
18  -spec handle_call(get_count, any(), {state,integer()}) ->
19    {reply, integer(), {state, integer()}}.
20
21  handle_call(get_count, _From, #state{count=Count}) ->
22    {reply, Count, #state{count=Count+1} }.
23
24  -spec handle_cast(stop | say_hello, {state,integer()}) ->
25    {stop, any(), {state, integer()}}
26     | {noreply, {state, integer()}}.
27
28  handle_cast(stop, State) ->
29      {stop, normal, State};
30
31  handle_cast(say_hello, State) ->
32      io:format("Hello~n"),
33      {noreply,
34        % #state{count = State#state.count+1} % RIGHT
35        #state{count = State#state.count-1} % WRONG
36      }.
```

FIGURE 10.24: `hello_server.erl`

```
 1  -module(test_hello_server).
 2  -export([rel/0, funs/0]).
 3  poi1Old() -> {'hello_server.erl', 22, tuple, 1}.
 4  poi2Old() -> {'hello_server.erl', 29, tuple, 1}.
 5  poi3Old() -> {'hello_server.erl', 33, tuple, 1}.
 6  poi1New() -> {'hello_server_wrong.erl', 22, tuple, 1}.
 7  poi2New() -> {'hello_server_wrong.erl', 29, tuple, 1}.
 8  poi3New() -> {'hello_server_wrong.erl', 33, tuple, 1}.
 9
10  funs() ->
11    "[handle_call/3, handle_cast/2]".
12
13  rel() ->
14    [{poi1Old(),poi1New()},{poi2Old(),poi2New()},{poi3Old(),poi3New()}].
```

FIGURE 10.25: Configuration file of `hello_server` programs

an error is reported in function `handle_cast`, pointing to the POI defined in line 5 of Figure 10.25. This means that for the request `say_hello` the behaviour has not been preserved, while for the request `stop` it has been preserved. In particular, the error found reveals that there is a discrepancy between the new server's state returned by each version of the program.

This simple example shown how SecEr can be used to check behaviour preservation even in concurrent context. The key is that there is no need to run an execution with real concurrency, instead we can study directly the relevant functions that are used during the concurrent execution, like `handle_call` or `handle_cast` in the example.

10.4 EDBC

Erlang design-by-contract (EDBC) is a library composed by a set of Erlang modules that allow users to include in their programs verification contracts like preconditions or postconditions to detect runtime incoherences. Compiling and

```
$ secer -pois "test_hello_server:rel()" -funs "test_hello_server:funs()"
        -to 15

Function: handle_call/3
----------------------------
Generated test cases: 19083
Both versions of the program generate the expected result for the defined
POIs
----------------------------

Function: handle_cast/2
----------------------------
Generated test cases: 42
Mismatching test cases: 21 (50.0%)
  Error Types:
    + different_value => 21 Errors
        Example call: handle_cast(say_hello,{state,4})

------ Detected Error ------
Call: handle_cast(say_hello,{state,4})
Error Type: different_value
- - - - - - - - - - - - - - -
Error Type: Unexpected trace value
POI: ({‘hello_server.erl’,33,tuple,1})
  Trace:
    [{noreply,{state,5}}]
POI: ({‘hello_server_wrong.erl’,33,tuple,1})
  Trace:
    [{noreply,{state,3}}]
----------------------------
```

FIGURE 10.26: SecEr reports discrepancies in the functions implementing the requests

executing the code using EDBC results in a code transformation that explicitly include in the code the set of conditions defined by the included contracts. During runtime all the contracts are evaluated for each contracted function call, providing a normal execution without notification or raising an error message in case a particular call violates any of the contracts. EDBC also include two new Erlang behaviours that enhance the `gen_server` Erlang behaviour functionality. These behaviours gives the user the capacity of automatically delaying those incoming requests that are incompatible with the server's state instead of just ignoring them.

https://github.com/serperu/edbc

10.4.1 Installation and first steps

To build and use EDBC in your computer, it is necessary to download the EDBC project from git.

Generate Docker testing environment

Once the git project has been downloaded, a docker container can be built by running the following commands from the edbc folder:

```
~$ docker build -t edbc .
~$ docker run --name edbc_container -it edbc bash
```

After running the second command, a new shell environment will be entered, and we can start using the tool by following the command guide explained in the following subsections.

Manual installation

When manually installing the tool, it is important to consider that all the Erlang/OTP dependences are already included in the modules of EDBC. For this reason, one of the requirements to correctly run EDBC is to have an Erlang distribution installed in your computer. Finally, a makefile is used to build the project, thus you need to have Make also installed in your computer.

```
~$ git clone https://github.com/serperu/edbc.git
~$ cd edbc
~$ make
```

First steps

The release version of EDBC contains a set of folders with Erlang files with their associated binary folders (ebin). It is important to keep these resources in the directory they have been generated, because the scripts follow this structure when charging Erlang modules. EDBC does not provide a GUI, it operates with a set of scripts launched from the terminal. EDBC is formed by four different scripts with their associated arguments, located in the `scripts` folder:

```
~$ edbc_erlc    FILE [OUTPUT_DIR]
~$ edbc_erlcp   FILE [OUTPUT_DIR]
~$ edbc_erl     EBIN_DIR [CALL]
~$ edbc_edoc    FILE EDOC_DIR
```

- `edbc_erlc/edbc_erlcp`
  - `FILE`. Represents the Erlang file (which usually includes contracts) that needs to be compiled with EDBC for the testing of some methods.
  - `OUTPUT_DIR`. Contains the (optional) output path where the `.beam` files generated when compiling the file of the first parameter will be stored.
- `edbc_erl`
  - `EBIN_DIR`. Indicates the directory where the `.beam` files that must be loaded when launching the Erlang terminal. Most times, this directory is the output directory of the `edbc_erlc` command, which contains the modules to be tested.

  - `CALL`. Contains a particular call to a loaded module that usually tests the contracted function. This parameter usually contains a function with a set of calls to contracted functions, which tries to discover potential inputs that violate any of the imposed contracts. If this argument is present, when the evaluation of the given call concludes the Erlang process is stopped. On the other hand, if the argument is missing, the Erlang process reamains open until it is manually stopped.

- `edbc_edoc`
  - `FILE`. Designate the Erlang file with contracted functions which documentation must be automatically obtained.
  - `EDOC_DIR`. Includes the output path where the documentation files (a set of `.html` files) must be saved.

In order to use EDBC to execute a module with defined contracts, we need to run two of these scripts. First of all, we need to compile the module using the `edbc_erlc` script, and then run `edbc_erl` to initiate an Erlang node that automatically loads the previously compiled module.

Example 10.6. *Consider the Erlang module in Figure 10.27 called `simple`, located in the path `./examples/src/simple.erl`. Module `simple` includes a function `f/1` with two contracts, a precondition contract and a postcondition contract. When we want to run different calls to this contracted function, first we need to compile it by executing the `edbc_erlc` script. Then, we can open an Erlang terminal with the `edbc_erl` script. The execution of the two mentioned scripts would be the following:*

```
~$ ./scripts/edbc_erlc ./examples/src/simple.erl ./examples/ebin
~$ ./scripts/edbc_erl ./examples/ebin
```

The result opens an Erlang terminal where the `simple` module has been already loaded. Now, all its functions can be freely called from this terminal. When we call function `f`, if any contract is violated during `f`'s execution, it will show an error message. For instance, the call `f(6)` violates the postcondition contract and generates the error shown in Figure 10.28.

The commands used to run Example 10.6 has been coded in the `Makefile` of EDBC. This particular example can be run by executing in the terminal the command: `~$ make run_simple`. Many other example cases, which are useful to understand how EDBC is used, have been also coded inside `Makefile`. In the next Section, we also use an example that can be also found there.

10.4.2 Use cases

Consider the Erlang module `sel_recv` shown in Figure 10.29, which implements the problem of selective receives explained in Section 9.1.2 of Chapter 9 with the `gen_server_cpre` behaviour implemented in EDBC. The program in this file, prints a message every time a request is delayed (line 27). Consider

```
-module(simple).
-export([f/1]).

-include_lib("edbc.hrl").

?PRE(fun pre_f/0).
f(0) -> 10;
f(N) ->
  Prev = f(N-1),
  Prev - 1.
?POST(fun post_f/0).

pre_f() ->
  case ?P(1) >= 0 of
    true ->
      true;
    false ->
      { false,
        "The first parameter should be "
        "greater than or equal to 0."}
  end.

post_f() ->
  io:format("f(~p) = ~p\n", [?P(1), ?R]),
  ?R >= ?P(1).
```

FIGURE 10.27: Module `simple` with pre- and post-condition contracts

```
> simple:f(6).
f(0) = 10
f(1) = 9
f(2) = 8
f(3) = 7
f(4) = 6
f(5) = 5
f(6) = 4
** exception error: "The postcondition does not hold. Last call: simple:f(6). Result: 4"
     in function edbc_lib:show_post_report/3 (src/edbc_lib.erl, line 185)
     in call from simple:f/1
     in call from simple:f1610/1
     in call from edbc_lib:post/2 (src/edbc_lib.erl, line 135)
     in call from simple:f/1
     in call from simple:f1610/1
     in call from edbc_lib:post/2 (src/edbc_lib.erl, line 135)
     in call from simple:f/1
```

FIGURE 10.28: Contract violation returned from call `simple:f(6)`

also the module `sel_recv_test` in Figure 10.30, that implements a simple test for the server. The test starts the server, calls to the `test` method of the server, and then stops it (note that the `stop` method contains a `timer:sleep/1` call to let the user observe how messages are delayed and served before halting the Erlang process).

In the `sel_recv`, the server implements a precondition to serve any synchronous request. This precondition is specified by the `cpre` function. Method `cpre` always returns a tuple of two elements, where the first element indicates

```
1  -module(sel_recv).
2  -behaviour(gen_server_cpre).
3  -include_lib("edbc.hrl").
4  -export([start_link/0, stop/0]).
5  -export([init/1, handle_call/3, handle_cast/2,
6    handle_info/2, cpre/3, terminate/2,
7    code_change/3]).
8  -export([test/0]).
9
10 start_link() ->
11   gen_server_cpre:start_link({local, ?MODULE},
12     ?MODULE, [], []).
13
14 stop() ->
15   timer:sleep(100),
16   gen_server_cpre:stop(?MODULE).
17
18 test() ->
19   gen_server_cpre:call(?MODULE, test).
20
21 init([]) ->
22   {ok, 0}.
23
24 cpre({result, N}, _, State = [N|R]) ->
25   {true, State};
26 cpre({result, N}, _, State) ->
27   io:format("Message ~p delayed",[N]),
28   {false, State};
29 cpre(test, _, State) ->
30   {true, State}.
31
32 handle_call(test, _From, _State) ->
33   List = [1,2,3,4,5,6,7,8,9],
34   lists:map(
35     fun(N) ->
36       spawn(fun() ->
37         gen_server_cpre:call(?MODULE,
38             {result, N})
39       end)
40     end,
41     lists:reverse(List)),
42   {reply, ok, List};
43 handle_call({result, N}, _From, [N|R]) ->
44   io:format("served: " ++
45      integer_to_list(N) ++ "\n"),
46   {reply, ok, R}.
47
48 handle_cast(_Request, State) ->
49   {noreply, State}.
50
51 handle_info(_Info, State) ->
52   {noreply, State}.
53
54 terminate(_Reason, _State) ->
55   ok.
56
57 code_change(_OldVsn, State, _Extra) ->
58   {ok, State}.
```

FIGURE 10.29: Server of the problem of the selective receives in Section 9.1.2 of Chapter 9

```
-module(sel_recv_test).
-export([test/0]).

test() ->
  sel_recv:start_link(),
  sel_recv:test(),
  sel_recv:stop(),
  ok.
```

FIGURE 10.30: Test for the problem presented with the `sel_recv` server of Figure 10.29

whether the request can be served or not, and the second one indicates the state of the server. In this case, according to the pattern matching of the `cpre` clause in line 24, the `{result,N}` request can only be served when `N` coincides with the first element of the list that indicates the state of the server (`[N|R]`). When this match is not given, the request is returned to the end of the server mailbox queue, for trying to solve it later.

To test this implementation, we execute these two commands:

```
~$ ./scripts/edbc_erlc   "examples/sel_recv/gen_server_qcpre/src/*.erl"
                         examples/sel_recv/ebin
~$ ./scripts/edbc_erl    examples/sel_recv/gen_server_qcpre/ebin
                         "sel_recv_test:test()"
```

This particular example can be run using the Makefile in the edbc root folder by executing the command: `~$ make test_sel_recv_q` in the terminal. The use of "*.erl" in the first command compiles all the modules in the specified folder, and the second one runs the `test` function of the `sel_recv_test` module. The result of this execution is shown in Figure 10.31, where it can be appreciated that some messages need to be delayed (sometimes several times) before being processed by a specific server state.

```
Message 9 delayed            Message 3 delayed            Message 9 delayed
Message 8 delayed            served: 2                    Message 8 delayed
Message 7 delayed            Message 9 delayed            Message 7 delayed
Message 6 delayed            Message 8 delayed            Message 6 delayed
Message 5 delayed            Message 7 delayed            served: 5
Message 4 delayed            Message 6 delayed            Message 9 delayed
Message 3 delayed            Message 5 delayed            Message 8 delayed
Message 2 delayed            Message 4 delayed            Message 7 delayed
served: 1                    served: 3                    served: 6
Message 9 delayed            Message 9 delayed            Message 9 delayed
Message 8 delayed            Message 8 delayed            Message 8 delayed
Message 7 delayed            Message 7 delayed            served: 7
Message 6 delayed            Message 6 delayed            Message 9 delayed
Message 5 delayed            Message 5 delayed            served: 8
Message 4 delayed            served: 4                    served: 9
```

FIGURE 10.31: Result of running `sel_recv_test:test()`

To sum up, while the `gen_server` behaviour does not allow the delay of requests received by a server, the use of `gen_server_cpre` gives the user an extra tool to serve all the requests, delaying those requests that cannot be served in a specific moment due to the state of the server. In the example, the result shows that all the messages are served in the expected order despite being sent in the inverse order (order selected by the `map` function of lines 34–41). This example and many other ones implementing the readers-writers problem, a semaphore problem, and some others can be found in the github repository: `https://github.com/serperu/edbc`.

Part V

Conclusions and Future Research

Chapter 11

Conclusions

The process of software maintenance starts from the moment our code is ready to be tested. In any real project, each fragment of code added to a software under construction is always followed by long testing and debugging sessions. These sessions go from the unitary tests used by the programmer during the development to the execution of structured test suites to assess the impact of every code change in the whole system. Additionally, the continuous changes in the system requirements sometimes introduce drastic code updates that require the application of new testing and debugging methods. Thus, in order to ensure the quality of the software, it is essential to include in the software production cycle effective and efficient debugging and testing mechanisms.

During the maintenance stage of any software project, corrective maintenance attracts significant attention since it is the one focused in fixing discovered problems and bringing software to an operational state for end users. Thus, this kind of maintenance usually has priority over other types of work [11]. Corrective maintenance completely depends on the detection of bugs with the use of complete testing methods, and on the location of these bug locations by means of advanced debugging methods.

Nowadays, many different types of testing techniques are used during software development and maintenance like model-based testing [46], fuzz testing [58, 119], concolic testing [113], or regression testing [185] among many others. Additionally, there exist different automatic or semi-automatic debugging techniques that guide the programmer during the search for the bug location in the code, e.g., abstract debugging [23], algorithmic debugging [178], or program slicing [204] to mention some of them.

In this thesis, we have addressed current state of the art problems present in some of these testing and debugging techniques, concretely program slicing, regression testing, and runtime verification. We highlight the main contributions of this thesis below.

- In Chapter 3, we specialised the definition of flow dependence in object-oriented (OO) programming [66]. First of all, considering the graph representation of the JSysDG, we have detected a problem in the representation of OO programs when object variables were selected as slicing criterion yielding the slicing process, in some cases, to incomplete slices. Then, after studying the nature of the error, we have defined a set of exclusive properties of object variables like partial and total definition, which prevent this kind of variables to be completely represented with the classic flow dependence definition. In consequence, after considering this set of properties, we have defined two new program dependences called object-flow dependence and object-reference dependence that accurately represent the connection between each object variable and both its data members and memory position, respectively. Additionally, we have proposed a new slicing algorithm that ensures the use of these dependences to guarantee completeness and improves precision keeping the linear-time performance of the standard slicing algorithm. The result of these contributions have increased the robustness of the JSysDG, improving the quality of the generated slices and solving the slice's completeness problem. Finally, we have implemented a new slicer for Java with the JSysDG in its basis (JavaSlicer) which includes object-flow and object-reference dependences. Our experimental evaluation has shown that incompleteness situations are frequently given in our benchmarks, generating an average slice increase of 26.69% (with an slowdown of approximately 1 millisecond) to achieve completeness in programs with more than one hundred nodes.

- In Chapter 4 we proposed a new approach for the representation and slicing of composite data structures, the Constrained-Edges Program Dependence Graph (CE-PDG) [61]. First of all, we augmented the PDG representation for explicit composite data structures by adding AST nodes representing inner components. Then, we added the new flow dependences between those components that appeared due to the decomposition of the data structures. After that, every edge was given a label containing a constraint used during the slicing process. Additionally, we defined a new slicing algorithm that takes profit of these constraints, limiting the access to some subcomponents of the data structures when traversing the graph, and enhancing the precision of the computed slices. This method is extended to interprocedural slicing by including in the process the generation of input, output, and summary edges and enhancing the CE-PDG to the CE-SDG. Finally, the representation model and the slicing algorithms have been implemented in a program slicer for Erlang called e-Knife, defined in Chapter 5. We have evaluated the impact of the proposed methodology in two different experiments: one for intraprocedural slicing and another for interprocedural slicing. In both experiments, the CE-SDG has shown a positive impact over the slice precision, reducing the size of the SDG slice in a 9.31% on average. We evaluate the results as positive, since the necessary overhead of achieving this considerable reduction is only a few milliseconds.

- Chapter 5 presented an alternative graph representation for programs, the Expression Dependence Graph (EDG), which can be used by any programming paradigm and language [63]. The EDG provides a fine-grained program representation, where each node corresponds with an AST node of the program instead of representing a program statement. As shown during this chapter, the EDG naturally represents the structure of many program statements that need special ad hoc transformations when being represented by the PDG. The decision of breaking down statements into multiple nodes as the AST representation suggests entails a new problem: the lack of a dependence connection between statement elements. To solve this problem, a new type of dependence called value dependence is defined between the components (expressions) inside a statement. Value dependence is language-dependent and determines whether an expression inside a statement is required to compute the value of another expression inside the same statement.

The key idea of the EDG is that all expressions are represented with two different nodes: the first one represents the syntactical representation of the expression in the program, and the second one represents the value of this expression. The chapter presented the process to build the EDG from the AST and compared the time to generate and slice both the PDG and EDG and the difference in precision of slicing both graphs. Finally, a program slicer based on the EDG model called e-Knife has been implemented for Erlang. The empirical evaluation performed comparing the behaviour of the SDG and the EDG when slicing Erlang programs (we used the program slicing benchmark suite Bencher, described in Chapter 6) has shown that, on average, the EDG is capable of reducing the slices computed by the SDG in a 14.20%.

- Chapter 6 exposed a methodology to compute quasi-minimal slices, which are minimal slices for a specific set of program inputs [152]. The chapter described a model that produces quasi-minimal slices by (optionally) using a group of static program slicers, and implementing the tree-ORBS slicing algorithm and a set of automatically generated test cases that attempt to maximise branch coverage. The contribution of this chapter is two-folded: first, the modelling and implementation of a minimal slice generator for Erlang, and second, the generation of Bencher, the first suite of Erlang programs with challenging slicing situations and their associated minimal slices, which are invaluable to measure the quality of any program slicer for Erlang. Finally, we have evaluated the performance of the methodology when generating the Bencher slicing suite. The application of tree-ORBS has been proved to be way more accurate that modern program slicers that work at AST level, like Slicerl and e-Knife, further reducing the result given by their combination in a 15.84% size of the original program on average. The experience obtained in our experimental evaluation suggests two relevant facts: the use of programs slicers in the first phase is not mandatory but improves the performance of the process; and, in the general case, the removal of only one node per iteration in tree-ORBS is enough to reach the minimal slice from a pair program-slicing criterion.

- In Chapter 8, we proposed a new testing approach, the Point Of Interest (POI) testing [87, 89, 153]. POI testing is used to compare the behaviour of an arbitrary program point in different program versions. In this approach, implemented for the Erlang programming language in a tool called SecEr, the user defines pairs of POIs from both different versions that are supposed to compute the same values. Given a list of POIs and a set of input functions, the program analyses these functions to generate a suite of test cases to automatically compare the pairs of POIs. Then, a set of transformation rules are applied over both programs to extract the values of the POIs as a side-effect when running each program. The cases where a pair of POIs generate mismatching results are detected and the user is provided with a report. Our tool SecEr, implemented in Erlang for Erlang is the first tool that implements this novel approach and, for this reason, it cannot be compared with any other tools based on POI testing. Thus, our experimental evaluation compared which of the different SecEr configurations used during the test case generation phase provides the best performance/time ratio. The results showed that, by far, the best generation process for most selected programs was the standalone test case mutation (where new test cases are obtained only recalculating some parameters of some previous test cases), improving the results obtained by concolic and random testing.

- Finally, Chapter 9 presented an implementation of the design-by-contract verification approach for Erlang [60]. This verification approach is implemented in a library for Erlang called EDBC. With the use of this library, the user can define contracts using a notation represented with Erlang macros. The library offers a set of seven different contracts (precondition, postcondition, invariants, time, purity...) for sequential and concurrent Erlang programs. All the defined contracts are evaluated during runtime, aborting the execution and reporting the user with an error message when any of them is violated. Additionally, EDBC defined new behaviours (`gen_server_cpre` and `gen_server_qcpre`) to resolve some specific situations in concurrent environments that use the `gen_server` behaviour in Erlang. The behaviours implement the possibility of delaying requests in a server considering the server inner state, annotating and classifying these requests with different priorities when needed. The EDBC library is implemented in Erlang for Erlang and all its contracts has been proved helpful in several practical situations described in this thesis.

In the field of static analysis, different analysis techniques rely on graphs to represent the whole set of dependences that exist between program statements. It is the case of most program slicing proposals, where a well-formed graph representation of the program is a key factor. Unfortunately, due to the continuous evolution of current programming languages, not all program statements can be trivially represented. In fact, although different programming languages contain similar constructions, they frequently need to be represented differently. This continued evolution is responsible for the permanent refinement of the existing graph representations, specially those used in program slicing. Despite that in some program slicing techniques it is acceptable to compute incomplete slices (e.g., those techniques where the slice needs to be computed for a particular set of inputs), in the general case, program slicers prioritise completeness over correctness. To this end, program slicers adopt a conservative position when dealing with difficult program structures, including in the slice the whole structure instead of making an elaborated dependence analysis over their components. In this thesis, we have proposed complementary and novel techniques to enhance the representation of some program structures used in different paradigms, in some cases detecting and solving completeness problems (proposing a set of missing graph dependences such as object-flow and object-reference dependences), and in others increasing the accuracy of the computed slices by improving the graph representation (with the proposal of the CE-SDG and the EDG graph representations).

Since it was proposed by Weiser in 1981, the area of program slicing has been highly explored topic, especially before the early 2000's. Nowadays, the interest on other trending topics, like artificial intelligence or big data among many others, has significantly reduced the proposals exclusively focused on program slicing. In most cases, these researches related to program slicing use some of the powerful dependence analysis used in program slicing (such as flow dependence or control dependence analyses) as a support tool to reach other goals. Program slicing has been studied for different paradigms and languages. On the one hand, most of the studies done in program slicing are conducted for imperative programming where different program slicers have been developed. Unfortunately, the amount of program slicers implemented for imperative programming (mostly for OO languages like Java or C++) is limited. Most program slicers publicly available are commonly academic prototypes stuck in past language versions (it is hardly possible to use them due to the evolution of the host language) or modules integrated into higher refactoring tools with strict installation requirements that difficult their use. There are also other private program slicers for imperative languages like CodeSurfer or CodeSonar, but their implementation is not accessible so we can only speculate about their internal models and architecture. On the other hand, if we look at the declarative paradigm, few are the program slicers proposed for logic and functional languages. In this case, the modelling of features used in declarative programming (pattern matching, high order, anonymous functions...) or the use of data structures considered as raw data types (lists and tuples) are hardly explored. Some proposals to represent these features and structures are proposed in the literature for declarative languages like Erlang or Haskell, but the dependence analysis and program modelling in these languages is an interesting study niche still unexplored.

In general, many program slicing papers draw overall ideas to deal with certain language features without describing them in detail. This is the case of how to represent some classic Java features like enums and generic classes, or some other features added to Java a few years ago, like anonymous functions or lambda expressions. This is reinforced when we deal with structures typically used in declarative programming, like list comprehensions, pattern matching, or higher order functions, where most lack an explicit representation proposal. The same happens with the static dependence analysis of complex scenarios like concurrency, where only a few papers that model the memory-shared concurrency scenario have been published.

Today, program slicing is commonly considered as part of other complex static analyses rather than a technique used in a standalone way for program analysis. To increase its application during the debugging process, it would be interesting to apply it to specific parts of the code like certain program methods or modules (instead of computing the whole model of a complete software system, which is highly time-consuming for large systems). To this end, it would be positive to propose models that accurately represent all the constructions of programming languages (considering languages of all programming paradigms) that are not specifically considered in the current graph representations. This is precisely what we did in this thesis, the proposals to represent different language features in Java and Erlang programs make the progress of our research to follow this line. As a result of our research we have defined new types of program dependences and new representations for several mentioned language structures. There is still much work left in the program slicing area and we are convinced that, if the research continues in this direction, we will incur in new program dependences yet unconsidered, which will enrich the current graph proposals and the program slicing applicability.

On the other side, in the area of testing and verification, this thesis contains two valuable contributions. The first one is the proposal of a new methodology called POI testing to detect software evolution bugs. The methodology can be used both to generate a suite of tests given a program and a function to be used as entry point and to compare two arbitrary points of two different code versions, which is the main novelty with respect to other approaches. The technique is promising, but the detection of behaviour differences depends on the generated suite of test cases, so the key factor to obtain a good result is the quality of the input generators. The computation of good inputs becomes more difficult when an input represents a complex data structure (e.g., a map or a sequence of nestled lists and tuples). Another current limitation is that the POIs need to be manually identified by the programmer, and this is sometimes difficult when a massive refactoring is performed over the code. POI testing is still in its first stages of development and, in its current state, its use is more suitable for detecting code behaviour changes in isolated functions or modules. There is still much work to do to make it suitable for being automatically used in large industrial projects with complex data types. The second proposal is an implementation for Erlang of the design-by-contract verification, giving support for precondition, postcondition and many other types of contracts to Erlang programmers. POI testing and contracts can be used in a symbiotic way to test critical functions by generating a suite of tests with the first one, and running the corresponding contracted function with the second one. This way, any contract violation would be detected during the execution of the tests and the error can be corrected before publishing a new release.

Chapter 12

Future Lines of Research

In this section, we discuss possible directions for future research along the lines of our work. We split the discussion in two categories corresponding to the main topics covered in this thesis.

12.1 Program Slicing

Although different program slicing situations have been analysed during this thesis, there are numerous related language features that have not been addressed yet in our approaches:

- **Object-Oriented Program Slicing**. There are some features related to Java OO programs which representation have not been modelled yet. Some of them are: the representation of inheritance and data members of enum types in Java, the dependence connection between static fields and static blocks inside the representation of a ClDG (initialised at runtime during the first reference to a class), the modelling of inner classes with their own data members and methods with limited visibility, or the representation of dynamic objects that locally override certain class methods, among many others. The modelling and implementation of some of the mentioned scenarios would result into a very interesting contribution for the area of static analysis and, in particular, for program slicing. Our implementation should also be improved by introducing a point-to analysis implementation, entering in a new discussion about the viability of context-sensitive vs context-insensitive point-to analysis. A suitable point-to analysis, together with the introduction of the object-flow and object-reference dependences would result in an immediate improvement of the computed slices, by providing an accurate detection of all object variables pointing to the same object when considering the definition and use sets.

- **Field-Sensitive Program Slicing**. With respect to the field-sensitive model proposed in the thesis, some scenarios are still to be considered. For example, the particular application of the proposed model to other potentially recursive data structures such as records or objects is not defined yet. The model has potential to represent data members of these constructs, but the technique used for it needs to be modified because their elements are commonly implicit when they are used in programs.

A mix between the constrained model and the OO approach used for Java in the JSysDG may be useful to represent these data structures, but the viability of this representation and the disadvantages it may suppose need to be explored yet. Another unsolved scenario is the appearance of structural dependences due to pattern matching. For instance, the tuple patterns in list comprehension generators requires the elements of the list to fulfil a particular data structure for the pattern matching to succeed. We are currently studying this apparently new concept of dependence and how to solve it by using the constrained model. Finally, we are currently implementing some ideas that would make the slicer generate some parts of the representation (e.g., the summary edges) only on demand.

- **Expression Dependence Graph**. Our implementation of the EDG is an initial prototype that has been proved to be viable to accurately represent programs of different languages. There exist endless possibilities to enhance the EDG: implementing the model for different programming languages, adapting it to work with the object-oriented representation models, implementing the exception-sensitive mechanisms, increasing the EDG with new complex data structures like records, or enhancing the EDG with many other techniques already proposed in the literature would significantly increase the potential of program slicing. Additionally, testing the potential of the EDG to model large systems with hundreds of modules would also be of great interest.

- **Quasi-minimal slicing**. QM-slicing is another area that can be potentially enhanced. Even though the technique is language-independent (it could be implemented to work in any programming language), our implementation is specific for Erlang. A language-independent implementation would be able to generate different program slicing suites to test and compare program slicers for different languages. Additionally, new slicing benchmarks could be added to the Bencher suite to deal with other complex structures like records or dictionaries.

- **Other program slicing fields**. Despite they are not addressed in this thesis, some other complex situations are worth to be studied in the area of program slicing. This is the case, e.g., of a definition of a static model to represent concurrency. Unfortunately, the study of concurrency has been less investigated. Some papers have identified a new kind of dependence called interference dependence.

As it can be appreciated, the field of program slicing grows in parallel with the computer science field. Every time a new programming language appears, a new representation model is necessary to statically analyse this language. The evolution of programming languages implies the development of new program representation techniques, that are closely related to the program slicing area. For this reason, the work in the area of program slicing is far from being closed soon.

12.2 Testing and Verification

In the testing and verification fields, our proposals are far from being finished. Different enhancements are yet to be theoretically and practically developed in both researches:

- **POI testing**. POI testing is an upcoming idea proposed in this thesis and it can be enhanced in several directions. Currently, the approach is only defined for Erlang, i.e., the transformation rules are applicable to Erlang. The first addition would be to generalise those rules to be applicable for a generic language instead of being restricted to Erlang. This process would require the definition of language-agnostic transformation rules and models where the analysis tools should be replaced depending on the programming language. This process may result in a set of rules like the ones defined in Section 8.3.2 for the same statements depending on language-related properties. Other directions for future development would be to automatise POI testing. This would involve the design of a process to automatically detect potential POIs using a diff analysis, together with the ability of automatically re-running the process by moving the POIs to certain alternative expressions when an unexpected behaviour is detected, automatically delimiting the source code where the error source is. Additionally, another interesting inclusion would be the use of other analysis techniques combined with POI testing (like program slicing) to reduce the amount of code being inspected when an unexpected behaviour is detected, preventing irrelevant parts of the code to be selected as POIs.

- **Erlang contracts**. The contracts library implemented for Erlang may include other types of contracts for functions such as those that ensure that a function that always ends raising an error, e.g., with the purpose to control those functions that are designed to fail. There are also some contracts with limited input possibilities that may be augmented. For instance, decreasing contracts can only compare if an integer argument decreases in successive recursive calls, but it cannot define a decreasing property over a non-integer argument itself, for example, EDBC cannot measure if the size of a list successively decreases between calls. EDBC can also be further enhanced with new types of contracts that may not work only at function level but at expression level. For instance, contracts to control that a `receive` expression in Erlang is always provided with messages that follow certain patterns, raising exceptions in case any unexpected pattern is received. Another interesting addition would be to include a contract capable of accessing parameters of previous recursive calls to verify any property that must be kept during the whole recursive process (e.g., when an element is removed from a list, it must never be found again inside that list).

POI testing and the Erlang design-by-contract models are currently usable in a controlled environment, but still have a lot of work to do to become mature tools applicable in industry. Some features should be included to make them more robust and give the programmer more instruments to control the quality of its development. In conclusion, POI testing is a novel powerful approach that has just been born with a large margin of improvement, and EDBC contains also a lot of unimplemented ideas that may result in very interesting approaches for runtime control of sequential and concurrent Erlang programs.

Bibliography

[1] Gagan Agrawal and Liang Guo. “Evaluating explicitly context-sensitive program slicing”. In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering.* 2001, pp. 6–12.

[2] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. “Dynamic slicing in the presence of unconstrained pointers”. In: *Proceedings of the symposium on Testing, Analysis, and Verification.* 1991, pp. 60–73.

[3] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition).* USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[4] Bernhard K. Aichernig. “Contract-based testing”. In: *Formal Methods at the Crossroads. From Panacea to Foundational Support.* Springer, 2003, pp. 34–48.

[5] Frances E. Allen. “Control Flow Analysis”. In: *SIGPLAN Not.* 5.7 (1970), pp. 1–19. ISSN: 0362-1340.

[6] 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.

[7] Jesus M. Almendros-Jimenez, Josep Silva, and Salvador Tamarit. “XQuery Optimization Based on Program Slicing”. In: *Proceedings of the 20th ACM International Conference on Information and Knowledge Management.* CIKM ’11. Glasgow, Scotland, UK: ACM, 2011, pp. 1525–1534. ISBN: 978-1-4503-0717-8.

[8] Saswat Anand et al. “An orchestrated survey of methodologies for automated software test case generation”. In: *Journal of Systems and Software* 86.8 (2013), pp. 1978–2001.

[9] 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.

[10] Sergio Antoy and Michael Hanus. “Contracts and Specifications for Functional Logic Programming”. In: *Practical Aspects of Declarative Languages - 14th International Symposium, PADL 2012, Philadelphia, PA, USA, January 23-24, 2012. Proceedings.* Ed. by Claudio V. Russo and Neng-Fa Zhou. Vol. 7149. Lecture Notes in Computer Science. Springer, 2012, pp. 33–47.

[11] Alain April and Alain Abran. “A software maintenance maturity model (S3M): Measurement practices at maturity levels 3 and 4”. In: *Electronic Notes in Theoretical Computer Science* 233 (2009), pp. 73–87.

[12] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[13] 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.

[14] Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Vol. 60. USA: Springer US, 1988.

[15] Benoit Baudry et al. “DSpot: Test Amplification for Automatic Assessment of Computational Diversity”. In: *CoRR* abs/1503.05807 (2015).

[16] David Binkley. “Precise Executable Interprocedural Slices”. In: *ACM Letters on Programming Languages and Systems* 2.1-4 (1993), pp. 31–45. ISSN: 1057-4514.

[17] David Binkley. “Slicing in the Presence of Parameter Aliasing”. In: *In Software Engineering Research Forum*. Orlando, FL, 1993, pp. 261–268.

[18] David Binkley and Keith B. Gallagher. “Program Slicing”. In: *Advances in Computers* 43.2 (1996), pp. 1–50.

[19] David Binkley, Nicolas Gold, and Mark Harman. “An Empirical Study of Static Program Slice Size”. In: *ACM Trans. Softw. Eng. Methodol.* 16.2 (2007), p. 8. ISSN: 1049-331X.

[20] 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.

[21] David Binkley et al. “Tree-Oriented vs. Line-Oriented Observation-Based Slicing”. In: *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2017, pp. 21–30.

[22] Barry W. Boehm. “A spiral model of software development and enhancement”. In: *Computer* 21 (1988), pp. 61–72.

[23] François Bourdoncle. “Abstract Debugging of Higher-Order Imperative Languages”. In: *ACM SIGPLAN Notices* 28.6 (1993), pp. 46–55. ISSN: 0362-1340.

[24] István Bozó et al. “Selecting Erlang test cases using impact analysis”. In: *AIP Conference Proceedings*. Vol. 1389. 1. AIP. 2011, pp. 802–805.

[25] Christopher M. Brown. “Tool Support for Refactoring Haskell Programs”. PhD thesis. School of Computing, University of Kent, Canterbury, Kent, UK, 2008.

[26] Rafael Caballero et al. “EDD: A Declarative Debugger for Sequential Erlang Programs”. In: *20th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. Springer, 2014, pp. 581–586.

[27] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, (OSDI 2008)*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 209–224.

[28] Richard Carlsson and Mickaël Rémond. “EUnit: a lightweight unit testing framework for Erlang”. In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*. Ed. by Marc Feeley and Philip W. Trinder. ACM, 2006, p. 1.

[29] Ian Cassar et al. “eAOP: an aspect oriented programming framework for Erlang”. In: *Proceedings of the 16th ACM SIGPLAN International Workshop on Erlang, Oxford, United Kingdom, September 3-9, 2017*. Ed. by Natalia Chechina and Scott Lystig Fritchie. ACM, 2017, pp. 20–30.

[30] Adnan Causevic, Daniel Sundmark, and Sasikumar Punnekkat. “An industrial survey on contemporary aspects of software testing”. In: *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE. 2010, pp. 393–401.

[31] Ned Chapin et al. “Types of software evolution and software maintenance”. In: *Journal of software maintenance and evolution: Research and Practice* 13.1 (2001), pp. 3–30.

[32] 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*. Elsevier ENTCS 177, 2007, pp. 123–136.

[33] Jiun-Liang Chen, Feng-Jian Wang, and Yung-Lin Chen. “Slicing object-oriented programs”. In: *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*. IEEE. 1997, pp. 395–404.

[34] Tsong Y. Chen and Y. Y. Cheung. “Dynamic program dicing”. In: *1993 Conference on Software Maintenance*. 1993, pp. 378–385.

[35] Yanping Chen, Robert L. Probert, and Hasan Ural. “Model-based regression test suite generation using dependence analysis”. In: *Proceedings of the 3rd Workshop on Advances in Model Based Testing, A-MOST 2007, co-located with the ISSTA 2007 International Symposium on Software Testing and Analysis, London, United Kingdom, July 9-12*. ACM, 2007, pp. 54–62.

[36] Zhenqiang Chen and Baowen Xu. “Slicing Concurrent Java Programs”. In: *SIGPLAN Not.* 36.4 (Apr. 2001), pp. 41–47. ISSN: 0362-1340.

[37] Zhenqiang Chen and Baowen Xu. “Slicing Object-Oriented Java Programs”. In: *SIGPLAN Not.* 36.4 (Apr. 2001), pp. 33–40. ISSN: 0362-1340.

[38] Pavan K. Chittimalli and Mary J. Harrold. “Recomputing coverage information to assist regression testing”. In: *IEEE Transactions on Software Engineering* 35.4 (2009), pp. 452–469.

[39] Jong-Deok Choi, Michael Burke, and Paul Carini. “Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects”. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1993, pp. 232–245.

[40] Edmund M. Clarke, Ernest A. Emerson, and Aravinda P. Sistla. “Automatic verification of finite-state concurrent systems using temporal logic specifications”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.2 (1986), pp. 244–263.

[41] Edmund M. Clarke et al. “Program slicing for VHDL”. In: *International Journal on Software Tools for Technology Transfer* 4.1 (2002), pp. 125–137.

[42] Holger Cleve and Andreas Zeller. “Finding Failure Causes through Automated Testing”. In: *Proceedings of the Fourth International Workshop on Automated Debugging*. Munich, Germany, 2000.

[43] Christian Colombo, Adrian Francalanza, and Rudolph Gatt. “Elarva: A Monitoring Tool for Erlang”. In: *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*. Ed. by Sarfraz Khurshid and Koushik Sen. Vol. 7186. Lecture Notes in Computer Science. Springer, 2011, pp. 370–374.

[44] Patrick Cousot and Radhia Cousot. “Inductive definitions, semantics and abstract interpretations”. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1992, pp. 83–94.

[45] Mats Cronqvist. *redbug*. Available at: https://github.com/massemanet/redbug. 2017.

[46] Siddhartha R. Dalal et al. “Model-based testing in practice”. In: *Proceedings of the 21st international conference on Software engineering*. 1999, pp. 285–294.

[47] Benjamin Danglot et al. “The Emerging Field of Test Amplification: A Survey”. In: *CoRR* abs/1705.10692 (2017).

[48] Andrea De Lucia et al. “Unions of Slices Are Not Slices”. In: *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*. CSMR ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 363–. ISBN: 0-7695-1902-4.

[49] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. “Critical Slicing for Software Fault Localization”. In: *SIGSOFT Softw. Eng. Notes* 21.3 (1996), pp. 121–134. ISSN: 0163-5948.

[50] Emelie Engström and Per Runeson. "A Qualitative Survey of Regression Testing Practices". In: *Product-Focused Software Process Improvement, 11th International Conference, PROFES 2010, Limerick, Ireland, June 21-23, 2010. Proceedings*. Ed. by Muhammad Ali Babar, Matias Vierimaa, and Markku Oivo. Vol. 6156. Lecture Notes in Business Information Processing. Springer, 2010, pp. 3–16. ISBN: 978-3-642-13791-4.

[51] Ericsson AB. *dbg*. Available at: `http://erlang.org/doc/man/dbg.html`. 2017.

[52] Ericsson AB. *EDoc*. Available at: `http://erlang.org/doc/apps/edoc/chapter.html`. 2018.

[53] Ericsson AB. *Trace Tool Builder*. Available at: `http://erlang.org/doc/apps/observer/ttb_ug.html`. 2017.

[54] *Erlang-Cover*. Available at: `http://www.erlang.org/doc/apps/tools/cover_chapter.html`. 1997.

[55] Michael D. Ernst. *Practical fine-grained static slicing of optimized code.* Tech. rep. Technical Report MSRTR-94-14, Microsoft Research, Redmond, WA, 1994.

[56] 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.

[57] John Field, Ganesan Ramalingam, and Frank Tip. "Parametric Program Slicing". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '95).* San Francisco, California, United States: ACM, 1995, pp. 379–392. ISBN: 0-89791-692-1.

[58] Daniel S. Fowler et al. "Fuzz testing for automotive cyber-security". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE. 2018, pp. 239–246.

[59] Lars-Åke Fredlund et al. "A testing-based approach to ensure the safety of shared resource concurrent systems". In: *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability* 230.5 (2016), pp. 457–472.

[60] Lars-Åke Fredlund et al. "Runtime Verification in Erlang by Using Contracts". In: *Functional and Constraint Logic Programming*. Ed. by Josep Silva. Cham: Springer International Publishing, 2019, pp. 56–73. ISBN: 978-3-030-16202-3.

[61] Carlos Galindo, Jens Krinke, and Sergio Pérez. "Field-Sensitive Program Slicing". In: *Software Engineering and Formal Methods: 20th International Conference, SEFM 2022, Berlin, Germany, September 26–30, 2022, Proceedings*. Vol. 13550. Springer Nature. 2022, pp. 74–90.

[62] Carlos Galindo, Sergio Pérez, and Josep Silva. "Conditional Control Dependence to Represent Catch Statements in the System Dependence Graph". In: *PROLE2021*. SISTEDES, 2021.

[63] Carlos Galindo, Sergio Pérez, and Josep Silva. "Fine-grained Graph Representation for Program Slicing (work in progress)". In: *PROLE2022.* SISTEDES, 2022.

[64] Carlos Galindo, Sergio Pérez, and Josep Silva. "Program Slicing with Exception Handling". In: *11th Workshop on Tools for Automatic Program Analysis.* 2020.

[65] Carlos Galindo, Sergio Pérez, and Josep Silva. "Slicing Unconditional Jumps with Unnecessary Control Dependencies". In: *Logic-Based Program Synthesis and Transformation.* Vol. 12561. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 293–308. ISBN: 978-3-030-68446-4.

[66] Carlos Galindo, Sergio Pérez, and Josep Silva. "Program slicing of Java programs". In: *Journal of Logical and Algebraic Methods in Programming* 130 (2022), p. 100826. ISSN: 2352-2208.

[67] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *SIGPLAN Not.* 42.10 (2007), pp. 57–76. ISSN: 0362-1340.

[68] Aggelos Giantsios, Nikolaos Papaspyrou, and Konstantinos Sagonas. "Concolic Testing for Functional Languages". In: *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP '15).* New York, NY, USA: ACM, 2015, pp. 137–148. ISBN: 978-1-4503-3516-4.

[69] Dennis Giffhorn and Christian Hammer. "An Evaluation of Slicing Algorithms for Concurrent Programs". In: *Proceedings of the 7th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM'07).* Maison Internationale, Paris: IEEE, 2007, pp. 17–26.

[70] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed automated random testing". In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation.* 2005, pp. 213–223.

[71] Xiaoyan Gongye et al. "A simple detection and generation algorithm for simple circuits in directed graph based on depth-first traversal". In: *Evolutionary Intelligence* (2020). ISSN: 1864-5917.

[72] Jurgen Graf. "Speeding Up Context-, Object- and Field-Sensitive SDG Generation". In: *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation.* 2010, pp. 105–114.

[73] Penny Grubb and Armstrong A. Takang. *Software maintenance: concepts and practice.* World Scientific, 2003.

[74] Ákos Hajnal and István Forgács. "A Demand-Driven Approach to Slicing Legacy COBOL Systems". In: *Journal of Software Maintenance* 24.1 (2012), pp. 67–82.

[75] Christian Hammer and Gregor Snelting. "An improved slicer for Java". In: *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering.* 2004, pp. 17–22.

[76] Christian Hammer and Gregor Snelting. “Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs”. In: *International Journal of Information Security* 8.6 (2009), pp. 399–422.

[77] Michael Hanus. “Combining Static and Dynamic Contract Checking for Curry”. In: *Proceedings of the 27th International Symposium on Logic-Based Program Synthesis and Transformation.* Vol. 10855. Lecture Notes in Computer Science. Springer, 2017, pp. 323–340.

[78] Ángel Herranz et al. “Modeling Concurrent Systems with Shared Resources”. In: *Formal Methods for Industrial Critical Systems, 14th International Workshop.* Vol. 5825. Lecture Notes in Computer Science. Springer, 2009, pp. 102–116.

[79] Carl Hewitt, Peter Bishop, and Richard Steiger. “A universal modular actor formalism for artificial intelligence”. In: *Proceedings of the 3rd international joint conference on Artificial intelligence.* 1973, pp. 235–245.

[80] Charles A. R. Hoare. “An axiomatic basis for computer programming”. In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[81] Tommy Hoffner. *Evaluation and Comparison of Program Slicing Tools.* Tech. rep. Sweden: LiTH-IDA-R-95-01 Department of Computer and Information Science, University of Kent, Linkping University: Sweden, 1995.

[82] Loïc Hoguin and William Dang. *Sheriff.* Available at: https://github.com/extend/sheriff. 2013.

[83] 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.

[84] 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.

[85] 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.

[86] “IEEE Standard for Software Test Documentation”. In: *IEEE Std 829-1998* (1998), pp. 1–64.

[87] David Insa et al. “Behaviour Preservation across Code Versions in Erlang”. In: *Scientific Programming* vol. 2018, Article ID 9251762 (2018), pp. 1–42.

[88] David Insa et al. “Erlang Code Evolution Control”. In: *Proceedings of the 27th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2017).* 2017.

[89] David Insa et al. “Erlang Code Evolution Control”. In: *Logic-Based Program Synthesis and Transformation (LOPSTR 2017).* Lecture Notes in Computer Science 10855 (2018), pp. 128–144.

[90] ISO. *ISO/IEC 14764 IEEE Std 14764-2006 Software Engineering - Software Life Cycle Processes - Maintenance.* Vol. 14764-2006. New York, NY, USA, 2006.

[91] Sonam Jain and Sandeep Poonia. “A New approach of program slicing: Mixed SD (static & dynamic) slicing”. In: *International Journal of Advanced Research in Computer and Communication Engineering Vol* 2 (2013).

[92] Shujuan Jiang et al. “Improving the Preciseness of Dependence Analysis Using Exception Analysis”. In: *2006 15th International Conference on Computing.* IEEE, 2006, pp. 277–282.

[93] Hao Jie, Jiang Shu-juan, and Hao Jie. “An approach of slicing for Object-Oriented language with exception handling”. In: *2011 International Conference on Mechatronic Science, Electric Engineering and Computer.* 2011, pp. 883–886.

[94] Miguel Jimenez, Tobias Lindahl, and Konstantinos Sagonas. “A language for specifying type contracts in Erlang and its interaction with success typings”. In: *Proceedings of the 2007 ACM SIGPLAN Workshop on Erlang, Freiburg, Germany, October 5, 2007.* Ed. by Simon J. Thompson and Lars-Åke Fredlund. ACM, 2007, pp. 11–17.

[95] Elroy Jumpertz. *Using QuickCheck and semantic analysis to verify correctness of Erlang refactoring transformations; Master’s thesis, Radboud University Nijmegen.* 2010.

[96] Yu Kashima, Takashi Ishio, and Katsuro Inoue. “Comparison of backward slicing techniques for java”. In: *IEICE TRANSACTIONS on Information and Systems* 98.1 (2015), pp. 119–130.

[97] 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.

[98] Bogdan Korel and Ali M. Al-Yami. “Automated regression test generation”. In: *ACM SIGSOFT Software Engineering Notes* 23.2 (1998), pp. 143–152.

[99] Bogdan Korel and Janusz Laski. “Dynamic program slicing”. In: *Information Processing Letters* 29.3 (1988), pp. 155 –163. ISSN: 0020-0190.

[100] 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.

[101] Gyula Kovács, Ferenc Magyar, and Tibor Gyimóthy. *Static Slicing of JAVA Programs.* Tech. rep. 96-108. Hungary: RGAI, Hungarian Academy of Sciences, Joesf Attila University, 1996.

[102] Herb Krasner. “The cost of poor software quality in the US: A 2020 report”. In: *Proc. Consortium Inf. Softw. QualityTM (CISQTM).* 2021.

[103] Jens Krinke. “Advanced Slicing of Sequential and Concurrent Programs”. PhD thesis. Universität Passau, 2003.

[104] Jens Krinke. “Context-Sensitive Slicing of Concurrent Programs”. In: *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 2003, 178–187.

[105] Jens Krinke. “Effects of context on program slicing”. In: *Journal of Systems and Software* 79.9 (2006), pp. 1249–1260.

[106] Jens Krinke. “Evaluating context-sensitive slicing and chopping”. In: *International Conference on Software Maintenance, 2002. Proceedings.* IEEE. 2002, pp. 22–31.

[107] 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.

[108] Prasanna Kumar et al. “A Static Slicing Method for Functional Programs and Its Incremental Version”. In: *Proceedings of the 28th International Conference on Compiler Construction.* CC 2019. Washington, DC, USA: Association for Computing Machinery, 2019, 53–64. ISBN: 9781450362771.

[109] 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. Springer, 2002, pp. 96–112. ISBN: 3-540-43353-8.

[110] Arun Lakhotia. *Improved Interprocedural Slicing Algorithm.* Tech. rep. CACS TR-92-5-8. Lafayette, LA 70504, USA: The Center for Advanced Computer Studies, University of Southwestern Louisiana, 1992.

[111] William Landi and Barbara G. Ryder. “A safe approximate algorithm for interprocedural aliasing”. In: *ACM SIGPLAN Notices* 27.7 (1992), pp. 235–248.

[112] 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.

[113] Eric Larson and Todd Austin. “High Coverage Detection of {Input-Related} Security Faults”. In: *12th USENIX Security Symposium.* 2003.

[114] K. Rustan M. Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16.* Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370.

[115] Huiqing Li and Simon Thompson. “Testing Erlang refactorings with QuickCheck”. In: *Symposium on Implementation and Application of Functional Languages*. Springer. 2007, pp. 19–36.

[116] Huiqing Li et al. “Refactoring Erlang Programs”. In: *Proceedings of the 12th International Erlang/OTP User Conference*. 2006.

[117] Huiqing Li et al. “Refactoring with Wrangler, Updated: Data and Process Refactorings, and Integration with Eclipse”. In: *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*. ERLANG ’08. Victoria, BC, Canada: ACM, 2008, pp. 61–72. ISBN: 978-1-60558-065-4.

[118] 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.

[119] Jie Liang et al. “Fuzz testing in practice: Obstacles and solutions”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2018, pp. 562–566.

[120] Tobias Lindahl and Konstantinos Sagonas. “Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story”. In: *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*. Vol. 3302. Lecture Notes in Computer Science. Springer, 2004, pp. 91–106. ISBN: 3-540-23724-0.

[121] Tobias Lindahl and Konstantinos Sagonas. “Practical type inference based on success typings”. In: *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*. 2006, pp. 167–178.

[122] Tobias Lindahl and Konstantinos Sagonas. “TypEr: a type annotator of Erlang code”. In: *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang, Tallinn, Estonia, September 26-28, 2005*. Ed. by Konstantinos Sagonas and Joe Armstrong. ACM, 2005, pp. 17–25.

[123] 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.

[124] Marisa Llorens et al. “Dynamic Slicing of Concurrent Specification Languages”. In: *Parallel Computing* 53 (2016), pp. 1–22. ISSN: 0167-8191.

[125] Marisa Llorens et al. “Dynamic Slicing Techniques for Petri Nets”. In: *Electronic Notes in Theoretical Computer Science* 223 (2006), pp. 153–165.

[126] David H. Lorenz and Therapon Skotiniotis. “Extending Design by Contract for Aspect-Oriented Programming”. In: *CoRR* abs/cs/0501070 (2005).

[127] Kasper Luckow et al. “JDart: A Dynamic Symbolic Analysis Framework”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Marsha Chechik and Jean-François Raskin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 442–459. ISBN: 978-3-662-49674-9.

[128] Wang Lulu, Li Bixin, and Kong Xianglong. “Type slicing: An accurate object oriented slicing based on sub-statement level dependence graph”. In: *Information and Software Technology* 127 (2020), p. 106369. ISSN: 0950-5849.

[129] James R. Lyle. “Evaluating Variations on Program Slicing for Debugging (Data-Flow, Ada)”. PhD thesis. USA, 1984.

[130] James R. Lyle and Mark Weiser. “Automatic Program Bug Location by Program Slicing”. In: *Proceedings of 2nd International Conference, Computers and Applications*. Vol. 2. Peking, China, 1987, pp. 877–883.

[131] Anirban Majumdar, Stephen J. Drape, and Clark D. Thomborson. “Slicing Obfuscations: Design, Correctness, and Evaluation”. In: *Proceedings of the 2007 ACM Workshop on Digital Rights Management*. DRM ’07. Alexandria, Virginia, USA: ACM, 2007, pp. 70–81. ISBN: 978-1-59593-884-8.

[132] Victor J. Marin and Carlos R. Rivero. “Towards a Framework for Generating Program Dependence Graphs from Source Code”. In: *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics*. SWAN 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 30–36. ISBN: 9781450360562.

[133] Dror Maydan, John Hennessy, and Monica Lam. “Efficient and Exact Data Dependence Analysis”. In: vol. 26. June 1991, pp. 1–14.

[134] Bertrand Meyer. “Applying "Design by Contract"”. In: *IEEE Computer* 25.10 (1992), pp. 40–51.

[135] Mehdi Mirzaaghaei. “Automatic test suite evolution”. In: *SIGSOFT/FSE’11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC’11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. Ed. by Tibor Gyimóthy and Andreas Zeller. ACM, 2011, pp. 396–399.

[136] Durga P. Mohapatra, Rajib Mall, and Rajeev Kumar. “An Overview of Slicing Techniques for Object-Oriented Programs”. In: *Informatica* 30.2 (2006), pp. 253–277.

[137] Melina Mongiovi. “Safira: A tool for evaluating behavior preservation”. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM. 2011, pp. 213–214.

[138] Steven S. Muchnick. *Advanced Compiler Design and Implementation, chapter 12.2*. Morgan Kaufmann, 1997.

[139] Steven S. Muchnick. *Advanced Compiler Design and Implementation, chapter 8.12*. Morgan Kaufmann, 1997.

[140] Steven S. Muchnick and Neil D. Jones. *Program flow analysis: Theory and applications*. Vol. 196. Prentice-Hall Englewood Cliffs, 1981.

[141] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[142] Leila Naslavsky, Hadar Ziv, and Debra J. Richardson. "MbSRT2: Model-Based Selective Regression Testing with Traceability". In: *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*. IEEE Computer Society, 2010, pp. 89–98.

[143] Hung V. Nguyen, Christian Kästner, and Tien N. Nguyen. "Cross-language program slicing for dynamic web applications". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 369–380.

[144] Claudio Ochoa, Josep Silva, and Germán Vidal. "Dynamic Slicing Based on Redex Trails". In: *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM '04)*. Verona, Italy: ACM, 2004, pp. 123–134. ISBN: 1-58113-835-0.

[145] Claudio Ochoa, Josep Silva, and Germán Vidal. "Lightweight Program Specialization via Dynamic Slicing". In: *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming*. New York, NY, USA: ACM, 2005, pp. 1–7. ISBN: 1-59593-069-8.

[146] Alessandro Orso, Saurabh Sinha, and Mary J. Harrold. "Effects of pointers on data dependences". In: *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*. IEEE. 2001, pp. 39–49.

[147] 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.

[148] Stack Overflow. *Stack Overflow Developer Survey 2021*. URL: https://insights.stackoverflow.com/survey/2021/ (visited on 04/11/2022).

[149] Carlos Pacheco et al. "Feedback-Directed Random Test Generation". In: *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 2007, pp. 75–84. ISBN: 0-7695-2828-7.

[150] Santosh K. Pani, Priya Arundhati, and Mahamaya Mohanty. "An Effective Methodology for Slicing C++ Programs". In: *International Journal of Computer Engineering and Technology* 1 (2010), pp. 72–82.

[151] Manolis Papadakis and Konstantinos Sagonas. "A PropEr integration of types and function specifications with property-based testing". In: *Proceedings of the 10th ACM SIGPLAN workshop on Erlang, Tokyo, Japan, September 23, 2011*. Ed. by Kenji Rikitake and Erik Stenman. ACM, 2011, pp. 39–50.

[152] Sergio Pérez, Josep Silva, and Salvador Tamarit. "Automatic Testing of Program Slicers". In: *Scientific Programming* vol. 2019, Article ID (2019), pp. 1–15.

[153] Sergio Pérez and Salvador Tamarit. “Enhancing POI Testing Through the Use of Additional Information”. In: *Functional and Constraint Logic Programming*. Ed. by Josep Silva. Cham: Springer International Publishing, 2019, pp. 74–90. ISBN: 978-3-030-16202-3.

[154] Keshav Pingali and Gianfranco Bilardi. “Optimal control dependence computation and the Roman chariots problem”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19.3 (1997), pp. 462–491.

[155] Mihalis Pitidis and Konstantinos Sagonas. “Purity in Erlang”. In: *Implementation and Application of Functional Languages - 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers*. Ed. by Jurriaan Hage and Marco T. Morazán. Vol. 6647. Lecture Notes in Computer Science. Springer, 2010, pp. 137–152.

[156] Hubert Plociniczak and Susan Eisenbach. “JErlang: Erlang with Joins”. In: *Coordination Models and Languages, 12th International Conference, COORDINATION 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings*. Ed. by Dave Clarke and Gul A. Agha. Vol. 6116. Lecture Notes in Computer Science. Springer, 2010, pp. 61–75.

[157] Krisztián Pócza, Mihály Biczó, and Zoltán Porkoláb. “Cross-language program slicing in the .NET framework”. In: *Proc. of the 3rd .NET Technologies Conference*. 2005, pp. 141–150.

[158] William Pugh and David Wonnacott. “Eliminating False Data Dependences using the Omega Test.” In: vol. 27. July 1992, pp. 140–151.

[159] Dawei Qi, Abhik Roychoudhury, and Zhenkai Liang. “Test generation to expose changes in evolving programs”. In: *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*. Ed. by Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto. ACM, 2010, pp. 397–406.

[160] Dawei Qi et al. “DARWIN: An approach to debugging evolving programs”. In: *ACM Trans. Softw. Eng. Methodol.* 21.3 (2012), 19:1–19:29.

[161] Jean-Pierre Queille and Joseph Sifakis. “Specification and verification of concurrent systems in CESAR”. In: *International Symposium on programming*. Springer. 1982, pp. 337–351.

[162] Jaspreet S. Rajal and Shivani Sharma. “Article: A Review on Various Techniques for Regression Testing and Test Case Prioritization”. In: *International Journal of Computer Applications* 116.16 (2015), pp. 8–13.

[163] Ganesan 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.

[164] Murali K. Ramanathan, Ananth Grama, and Suresh Jagannathan. “Sieve: A Tool for Automatically Detecting Variations Across Program Versions”. In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan.* IEEE Computer Society, 2006, pp. 241–252. ISBN: 0-7695-2579-2.

[165] Venkatesh-Prasad Ranganath. *Indus, a toolkit to customize and adapt Java programs.* Available at: http://indus.projects.cis.ksu.edu.

[166] Davi de Castro Reis et al. “Automatic Web News Extraction Using Tree Edit Distance”. In: *Proceedings of the 13th International Conference on World Wide Web (WWW'04).* New York, NY, USA: ACM, 2004, pp. 502–511. ISBN: 1-58113-844-X.

[167] 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.

[168] Thomas Reps and Todd Turnidge. “Program Specialization Via Program Slicing”. In: *Proceedings of the Dagstuhl Seminar on Partial Evaluation.* Vol. 1110. Lecture Notes in Computer Science. Springer-Verlag, 1996, pp. 409–429.

[169] Thomas Reps et al. “Speeding Up Slicing”. In: *SIGSOFT Softw. Eng. Notes* 19.5 (1994), pp. 11–20.

[170] Thomas Reps et al. “The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem”. In: *Software Engineering - ESEC/FSE '97, 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering, Zurich, Switzerland, September 22-25, 1997, Proceedings.* Ed. by Mehdi Jazayeri and Helmut Schauer. Vol. 1301. Lecture Notes in Computer Science. Springer, 1997, pp. 432–449.

[171] Nuno F. Rodrigues and Luís S. Barbosa. “Component Identification Through Program Slicing”. In: *In Proc. of Formal Aspects of Component Software (FACS 2005). Elsevier ENTCS.* Elsevier, 2005, pp. 291–304.

[172] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. “Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values”. In: *SIGPLAN Not.* 44.2 (2008), pp. 37–48. ISSN: 0362-1340.

[173] Per Runeson. “A Survey of Unit Testing Practices”. In: *IEEE Software* 23 (July 2006).

[174] Per Runeson, Carina Andersson, and Martin Höst. “Test processes in software product evolution: A qualitative survey on the state of practice”. In: *Journal of Software Maintenance* 15 (Jan. 2003), pp. 41–59.

[175] Konstantinos Sagonas, Josep Silva, and Salvador Tamarit. “Precise Explanation of Success Typing Errors”. In: *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*. PEPM ’13. Rome, Italy: ACM, 2013, pp. 33–42. ISBN: 978-1-4503-1842-6.

[176] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A concolic unit testing engine for C”. In: *ACM SIGSOFT Software Engineering Notes* 30.5 (2005), pp. 263–272.

[177] Ehud Y. Shapiro. *Algorithmic program debugging*. MIT Press, 1982.

[178] Josep Silva. “A Survey on Algorithmic Debugging Strategies”. In: *Advances in Engineering Software* 42.11 (2011), pp. 976–991. ISSN: 0965-9978.

[179] Josep Silva. “A Vocabulary of Program Slicing-Based Techniques”. In: *ACM Computing Surveys* 44.3 (2012).

[180] Josep Silva, Salvador Tamarit, and César Tomás. “System Dependence Graphs in Sequential Erlang”. In: *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012)*. Vol. 7212. Lecture Notes in Computer Science. Springer, 2012, pp. 486–500. ISBN: 978-3-642-28871-5.

[181] Josep Silva et al. *Slicerl*. 2011. URL: http://kaz.dsic.upv.es/slicerl.

[182] Anthony M. Sloane and Jason Holdsworth. “Beyond traditional program slicing”. In: *ACM SIGSOFT Software Engineering Notes*. Vol. 21. 3. ACM. 1996, pp. 180–186.

[183] Gregor Snelting, Torsten Robschink, and Jens Krinke. “Efficient Path Conditions in Dependence Graphs for Software Safety Analysis”. In: *ACM Trans. Softw. Eng. Methodol.* 15.4 (Oct. 2006), 410–457. ISSN: 1049-331X.

[184] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. “Automated behavioral testing of refactoring engines”. In: *IEEE Transactions on Software Engineering* 39.2 (2013), pp. 147–162.

[185] Software and Committee Systems. “IEEE Standard for Software and System Test Documentation”. In: *IEEE Std 829-2008* (July 2008), pp. 1 –118.

[186] 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 (Jan. 2019).

[187] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. “Thin Slicing”. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07. San Diego, California, USA: Association for Computing Machinery, 2007, 112–122. ISBN: 9781595936332.

[188] Christoph Steindl. “Intermodular slicing of object-oriented programs”. In: *Compiler Construction*. Ed. by Kai Koskimies. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 264–278. ISBN: 978-3-540-69724-4.

[189] E. Burton Swanson. “The Dimensions of Maintenance”. In: *Proceedings of the 2nd International Conference on Software Engineering*. ICSE ’76. San Francisco, California, USA: IEEE Computer Society Press, 1976, 492–497.

[190] Kuo C. Tai. “The Tree-to-Tree Correction Problem”. In: *Journal of the ACM* 26.3 (1979), pp. 422–433. ISSN: 0004-5411.

[191] Kunal Taneja and Tao Xie. “DiffGen: Automated Regression Unit-Test Generation”. In: *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L’Aquila, Italy*. IEEE Computer Society, 2008, pp. 407–410.

[192] Kunal Taneja et al. “eXpress: guided path exploration for efficient regression test generation”. In: *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*. Ed. by Matthew B. Dwyer and Frank Tip. ACM, 2011, pp. 1–11.

[193] Ramsay Taylor et al. “Using behaviour inference to optimise regression test sets”. In: *IFIP International Conference on Testing Software and Systems*. Springer. 2012, pp. 184–199.

[194] Andy Till. *erlyberly*. Available at: https://github.com/andytill/erlyberly. 2017.

[195] Frank Tip. “A Survey of Program Slicing Techniques”. In: *Journal of Programming Languages* 3.3 (1995), pp. 121–189.

[196] Paolo Tonella et al. “Flow insensitive C++ pointers and polymorphism analysis and its application to slicing”. In: *Proceedings of the 19th international conference on Software engineering*. 1997, pp. 433–443.

[197] Melinda Tóth and Zoltán Horváth. “Reduction of regression tests for Erlang based on impact analysis”. In: (2013).

[198] Melinda Tóth et al. “Impact analysis of erlang programs using behaviour dependency graphs”. In: *Proceedings of the Third summer school conference on Central European functional programming school*. CEFP’09. Budapest, Hungary: Springer-Verlag, 2010, pp. 372–390. ISBN: 3-642-17684-4, 978-3-642-17684-5. URL: http://dl.acm.org/citation.cfm?id=1939128.1939139.

[199] Melinda Tóth et al. “Impact analysis of Erlang programs using behaviour dependency graphs”. In: *Central European Functional Programming School*. Springer, 2010, pp. 372–390.

[200] Ivan Vankov. “Relational approach to program slicing”. In: *Master’s thesis, University of Amsterdam* (2005).

[201] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall International (UK) Ltd., 1996.

[202] 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.

[203] Daniel Weise et al. “Value Dependence Graphs: Representation without Taxation”. In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’94. Portland, Oregon, USA: Association for Computing Machinery, 1994, pp. 297–310. ISBN: 0897916360.

[204] Mark Weiser. “Program Slicing”. In: *Proceedings of the 5th international conference on Software engineering (ICSE ’81)*. San Diego, California, United States: IEEE Press, 1981, pp. 439–449. ISBN: 0-89791-146-6.

[205] Manfred Widera. “Flow graphs for testing sequential erlang programs”. In: *Proceedings of the 2004 ACM SIGPLAN workshop on Erlang*. ERLANG ’04. Snowbird, Utah, USA: ACM, 2004, pp. 48–53. ISBN: 1-58113-918-7.

[206] Manfred Widera and Fachbereich Informatik. “Concurrent Erlang flow graphs”. In: *In Proceedings of the Erlang/OTP User Conference 2005*. 2005.

[207] wired.co.uk. *Ubisoft is using AI to catch bugs in games before devs make them*. Available at: `https://www.wired.co.uk/article/ubisoft-commit-assist-ai`. 2018.

[208] Tao Xie and David Notkin. “Checking Inside the Black Box: Regression Testing by Comparing Value Spectra”. In: *IEEE Trans. Software Eng.* 31.10 (2005), pp. 869–883.

[209] Zhaogui Xu et al. “Static Slicing for Python First-Class Objects”. In: *2013 13th International Conference on Quality Software*. 2013, pp. 117–124.

[210] Shin Yoo and Mark Harman. “Regression Testing Minimization, Selection and Prioritization: A Survey”. In: *Softw. Test. Verif. Reliab.* 22.2 (2012), pp. 67–120. ISSN: 0960-0833.

[211] Kai Yu et al. “Practical isolation of failure-inducing changes for debugging regression faults”. In: *Proceedings of the 27th IEEE/ACM Int. Conference on Automated Software Engineering*. ACM. 2012, pp. 20–29.

[212] Andreas Zeller and Ralf Hildebrandt. “Simplifying and Isolating Failure-Inducing Input”. In: *IEEE Trans. Softw. Eng.* 28.2 (2002), pp. 183–200. ISSN: 0098-5589.

[213] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. “Injecting mechanical faults to localize developer faults for evolving software”. In: *ACM SIGPLAN Notices*. Vol. 48. 10. ACM. 2013, pp. 765–784.

[214] Zhihao Zhang et al. “Regression Test Generation Approach Based on Tree-Structured Analysis”. In: *Prodeedings of the 2010 International Conference on Computational Science and Its Applications, ICCSA 2010, Fukuoka, Japan, March 23-26, 2010*. Ed. by Bernady O. Apduhan et al. IEEE Computer Society, 2010, pp. 244–249.

[215] Jianjun Zhao. “Applying Program Dependence Analysis To Java Software”. In: *Proceedings of Workshop on Software Engineering and Database Systems*. Dec. 1998, pp. 162–169.

[216] Jianjun Zhao. “Slicing Aspect-Oriented Software”. In: *Proceedings of the 10th International Workshop on Program Comprehension*. IWPC ’02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 251–260. ISBN: 0-7695-1495-2.

Part VI

Appendices

Appendix A

Scientific Contributions

This appendix summarises all the contributions related to this thesis where the author has actively participated. The personal contributions of the author in all the mentioned research papers are the following: the author actively participated in the brainstorming sessions where the models and algorithms were defined, collaborated in the definition and proof of formal aspects like theorems and lemmas, and was an active part in the implementation and empirical evaluation of all the tools described in every paper.

A.1 Conference papers

- Carlos Galindo, Jens Krinke, Sergio Pérez, Josep Silva. A program slicer for Java (tool paper). 20th International Conference on Software Engineering and Formal Methods (SEFM 2022). Springer LNCS 13550, pages 146-151, 2022.

- Carlos Galindo, Jens Krinke, Sergio Pérez, Josep Silva. Field-Sensitive Program Slicing. 20th International Conference on Software Engineering and Formal Methods (SEFM 2022). Springer LNCS 13550, pages 74-90, 2022.

- Lars-Åke Fredlund, Julio Mariño, Sergio Pérez and Salvador Tamarit. Runtime verification in Erlang by using contracts. 26th International Workshop on Functional and Logic Programming (WFLP 2018). Springer LNCS 11285, pages 56-73, 2018.

- Sergio Pérez, Josep Silva, Salvador Tamarit. Enhancing POI testing approach through the use of additional information. 26th International Workshop on Functional and Logic Programming (WFLP 2018). Springer LNCS 11285, pages 74-90, 2018.

- David Insa, Sergio Pérez, Josep Silva, Salvador Tamarit. Erlang Code Evolution Control. 27th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2017). Springer LNCS 10855, pages 128-144, 2018.

- David Insa, Sergio Pérez, Josep Silva. Computing super reduced program slices by composing slicing techniques. International ACM Symposium on Applied Computing (SAC 2017). ACM 2017: 1631-1633.

- Carlos Galindo, Sergio Pérez, Josep Silva. XX Jornadas de Programación y Lenguajes (PROLE 2021). Object Variable Dependencies in Object-Oriented Programs. Digital library SISTEDES, 2021.
- David Insa, Sergio Pérez, Josep Silva, Salvador Tamarit. XVIII Jornadas de Programación y Lenguajes (PROLE 2018). Behaviour Preservation across Code Versions in Erlang. Digital library SISTEDES, 2018.
- David Insa, Sergio Pérez, Josep Silva. How to Construct a Suite of Program Slices. Proceedings of XVI Jornadas sobre Programación y Lenguajes (PROLE 2016). Proceedings of PROLE 2016: 200-215.

A.2 Journal Publications

- Carlos Galindo, Sergio Pérez, Josep Silva. Program Slicing of Java Programs. Journal of Logical and Algebraic Methods in Programming Vol. 130: 1-18, 2022.
- Sergio Pérez, Josep Silva, Salvador Tamarit. Automatic Testing of Program Slicers. Scientific Programming 4108652:1-15, 2019.
- David Insa, Sergio Pérez, Josep Silva, Salvador Tamarit. Behaviour Preservation across Code Versions in Erlang. Scientific Programming 9251762:1-42, 2018.

A.3 List of derived artifacts

Resource Name	**Type**	**URL**
Bencher	Web page	`https://mist.dsic.upv.es/bencher/`
JavaSlicer	Tool repository	`https://github.com/mistupv/JavaSlicer`
	Online version	`https://mist.dsic.upv.es/JavaSlicer/demo`
e-Knife	Tool repository	`https://mist.dsic.upv.es/git/program-slicing/e-knife-erlang`
	Online version	`https://mist.dsic.upv.es/e-knife-constrained/`
SecEr	Tool repository	`https://github.com/mistupv/secer`
EDBC	Tool repository	`https://github.com/serperu/edbc`