ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Optimization Techniques for Algorithmic Debugging

David Insa Cabrera

Supervised by:
Josep Silva Galiana

A Thesis presented for the degree of
Doctor of Philosophy at the Technical University of Valencia

June 2016

Optimization Techniques for Algorithmic Debugging

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

David Insa Cabrera

Supervisor

Josep Silva Galiana	Universidad Politécnica de Valencia

Reviewers

Olaf Chitil	University of Kent
Narciso Marti Oliet	Universidad Complutense de Madrid
Andreas Zeller	University of Saarland

Examiners

María Alpuente Frasnedo	Universidad Politécnica de Valencia
Rafael Caballero Roldán	Universidad Complutense de Madrid
Olaf Chitil	University of Kent

To mum and dad, because they always
went out of their way to protect and guide us.

Agradecimientos

Cuando terminé la carrera tomé una decisión que cambiaría el rumbo de mi vida, decidí convertirme en doctor. Mucho tiempo ha pasado desde entonces, y no han sido pocas las piedras que me he encontrado por el camino: algunas noches sin dormir, laberintos de los que no podía salir, trabajos que no sabía cómo completar, y montañas y montañas de tareas que nunca se acababan. En definitiva, no sabía dónde me estaba metiendo. A pesar de todo ello, esta ha sido una gran etapa de mi vida que si volviera atrás volvería a emprender, donde he aprendido mucho y he madurado como persona. No puedo decir que lo haya conseguido solo, todo esto se lo debo en gran medida a mucha gente que ha ayudado a que ahora pueda decir con orgullo *soy doctor*.

A las primeras personas a las que quiero agradecérselo es a aquellas que siempre han estado ahí, a mis hermanos Javier y Natalia. Con quienes siempre he podido contar, y quienes me han ayudado cuando he tenido algún problema. Quienes me han sabido dar los mejores consejos y han creído en que podía llegar a lo más alto. Habéis sido un pilar fundamental en mi vida y os agradezco mucho todo el apoyo incondicional que me habéis dado.

Una especial mención a quienes me dieron la vida e hicieron que pudiera vivir en este mundo, a mis padres. Disteis lo mejor de vosotros para aguantarme y soportarme, pero también para enseñarme, protegerme y guiarme a lo largo de toda mi vida. Siempre habéis procurado lo mejor para nosotros tres, nos habéis dado buenos consejos, pusisteis vuestro cuerpo y alma en que no nos faltara de nada y en que nos fuera lo mejor posible. Por eso y mucho más, este libro va dedicado a vosotros.

Gracias también a todas aquellas personas que no han tenido un impacto directo en esta etapa de mi vida, pero que aún así han estado ahí durante todo este tiempo. Me refiero a mi familia y amigos. Siempre me habéis apoyado, os habéis preocupado por cómo iban transcurriendo estos años y no habéis dudado en echarme una mano cuando ha sido necesario.

Si he realizado este doctorado ha sido gracias a quien me ha aceptado en su grupo de investigación. Alguien que ha sabido escuchar mis inquietudes y necesidades. Quien ha sabido gestionar los recursos de los que disponíamos para que nunca nos faltase nada. Gracias Germán por hacer todo lo posible para que todos nos sintiéramos parte de esta pequeña familia.

Quería hacer especial mención a una gran persona. Alguien que siempre hizo lo que estuviera en su mano para enseñarme y guiarme en el mundo de la investigación. Sin su ayuda, hoy en día no sería el mismo. Una persona que, por mucho que me duela reconocerlo, es una fuente de inspiración y sabiduría, a la vez que un gran jefe y amigo. Josep (iusep, iuseppe, iussepe, etc.), quería darte las gracias por todos estos años que me has dedicado. Me has enseñado a trabajar en equipo, a esforzarme siempre un poco más y a no rendirme. Durante estos últimos años no solo me has formado como investigador, sino también como persona. Por cierto, nunca nadie antes me había despedido tantas veces, ni tampoco me había intentado asesinar a sangre fría …

Durante los primeros años de mi doctorado compartí laboratorio con dos compañeros y amigos. Salvador Tamarit (Tama) y César fuisteis las primeras personas con las que trabajé mano a mano, y con vosotros descubrí que un trabajo no tiene por qué ser ni estresante ni aburrido. Gracias por todas las risas que nos hemos echado durante esos años, pero sobre todo aquellas que nos echamos en reuniones y conferencias (muchas de ellas a costa de Josep). Ha sido un placer haber compartido con vosotros los

primeros años de esta difícil etapa.

A mediados de mi doctorado alguien se infiltró en mi laboratorio, y ni estaba invitada ni se le pedía que volviese. Sin embargo, día tras día ahí estaba y, a pesar de ello, tengo algo que decirte. Laura Titolo (la italiana) gracias por haberme hecho compañía durante tantos meses. Gracias por amenizar las jornadas de trabajo con nuestras charlas interminables. Desde que te fuiste siento como si algo faltase en el laboratorio. Es una pena que nuestros caminos se hayan separado, te deseo lo mejor allí donde vayas.

Adrián y Sergio, fuisteis los últimos en llegar pero no por ello sois menos importantes. Durante las vacas flacas mucha gente se fue yendo del DSIC y poco a poco fui viendo cómo me quedaba solo en el laboratorio. Un día por suerte llegasteis vosotros y le disteis vida de nuevo a estas cuatro paredes. Gracias por volver a llenar de risas y buen rollo este nuestro laboratorio, y por aquellos descansos (más bien largos) para ir a tomarnos un café.

Durante mi doctorado hice dos estancias de investigación. La primera de ellas fue en Saarbrücken (Alemania) donde Andreas Zeller me acogió en el seno de su grupo de investigación. Allí trabajé con Juan Pablo Galeotti (el argentino) que me ayudó mucho durante mi estancia allí. No tenías por qué, pero ahí estuviste cuando lo necesité, muchas gracias.

Mi segunda estancia fue en Londres, en el grupo de Jens Krinke. Tengo que agradecerle el haber dedicado parte de su tiempo en reunirse y trabajar conmigo, así como recomendarme qué hacer en Londres. También conocí allí a Héctor (el madrileño), quien me ha enseñado y con quien me he pateado todo Londres. Siempre es de agradecer escuchar una voz amiga tan lejos de tu hogar. Me alegro mucho de haberte conocido, de haber pasado tres meses charrando y conspirando sobre cómo destruir Londres desde dentro. Echaré de menos ir a tomarnos cafés día tras día. Por cierto, espero que sigas teniendo la taza que te regalé :). Finalmente dar las gracias a Hanna (pounchus), la mujer más simpática y alegre que he conocido en mi vida. Me alegraste durante mi estancia y apoyaste durante los tiempos más difíciles. Gracias por ser como eres y por haber cambiado mi vida a partir de entonces.

También hay doctorandos y nuevas generaciones del grupo ELP de los que no me gustaría olvidarme. Entre ellos se encuentran Sonia, Marco, Francisco, Fernando Martínez, Fernando Tarín, Julián, Adolfo, Antonio, Julia, Ángel, Lidia, Daniel y Santiago. A Santiago lo incluyo porque ha pasado tanto tiempo entre nosotros que yo lo considero uno más. Chicos, ¡mucho ánimo! sobre todo a los que aún estáis luchando para terminar el doctorado. Por último, quiero acordarme del resto de profesores del MiST y ELP, grandes investigadores (cada uno en su campo) que siempre que lo he necesitado me han ayudado en lo que me hiciera falta.

Abstract

Nowadays, undetected programming bugs produce a waste of billions of dollars per year to private and public companies and institutions. In spite of this, no significant advances in the debugging area that help developers along the software development process have been achieved yet. In fact, the same debugging techniques that were used 20 years ago are still being used now. Along the time, some alternatives have appeared, but there still is a long way for them to be useful enough to get into the software development process. One of them is algorithmic debugging, which abstracts the information the user has to investigate to debug the program, allowing them to focus on what is happening instead of how it is happening. This abstraction comes at a price: the granularity level of the bugs that can be detected allows for isolating wrongly implemented functions, but which part of them contains the bug cannot be found out yet. This thesis focusses on improving algorithmic debugging in many aspects. Concretely, the main aims of this thesis are to reduce the time the user needs to detect a programming bug as well as to provide the user with more detailed information about where the bug is located. To achieve these goals, some techniques have been developed to start the debugging sessions as soon as possible, to reduce the number of questions the user is going to be asked, and to augment the granularity level of those bugs that algorithmic debugging can detect, allowing the debugger in this way to keep looking for bugs even inside functions. As a result of this thesis, three completely new techniques have been defined, an already existent technique has been improved, and two new algorithmic debugging search strategies have been defined that improve the already existent ones. Besides these theoretical results, a fully functional algorithmic debugger has been implemented that contains and supports all these techniques and strategies. This debugger is written in Java, and it debugs Java code. The election of this language is justified because it is currently one of the most widely extended and used languages. Also because it contains an interesting combination of unsolved challenges for algorithmic debugging. To further increase its usability, the debugger has been later adapted as an Eclipse plugin, so it could be used by a wider number of users. These two debuggers are publicly available, so any interested person can access them and continue with the research if they wish so.

Resumen

Hoy en día, los errores no detectados de programación suponen un gasto de miles de millones al año para las empresas e instituciones públicas y privadas. A pesar de esto, no ha habido ningún avance significativo en el área de la depuración que ayude a los desarrolladores durante la fase de desarrollo de software. De hecho, las mismas técnicas de depuración que se utilizaban hace 20 años se siguen utilizando ahora. A lo largo del tiempo, han surgido algunas alternativas, pero todavía queda un largo camino para que estas sean lo suficientemente útiles como para abrirse camino en el proceso de desarrollo de software. Una de ellas es la depuración algorítmica, la cual abstrae la información que el programador debe investigar para depurar el programa, permitiéndole de este modo centrarse en el qué está ocurriendo en vez de en el cómo. Esta abstracción tiene un coste: el nivel de granularidad de los errores que pueden detectarse nos permite como máximo aislar funciones mal implementadas, pero no averiguar qué parte de estas contiene el error. Esta tesis se centra en mejorar la depuración algorítmica en muchos aspectos. Concretamente, los principales objetivos de esta tesis son reducir el tiempo que el usuario necesita para detectar un error de programación así como proporcionar información más detallada de dónde se encuentra el error. Para conseguir estos objetivos, se han desarrollado técnicas para iniciar las sesiones de depuración lo antes posible, reducir el número de preguntas que se le van a realizar al usuario, y aumentar el nivel de granularidad de los errores que la depuración algorítmica puede detectar, permitiendo así seguir buscando el error incluso dentro de las funciones. Como resultado de esta tesis, se han definido tres técnicas completamente nuevas, se ha mejorado una técnica ya existente, y se han definido dos nuevas estrategias de depuración algorítmica que mejoran las previamente existentes. Además de los resultados teóricos, también se ha desarrollado un depurador algorítmico completamente funcional que contiene y respalda todas estas técnicas y estrategias. Este depurador está escrito en Java y depura código Java. La elección de este lenguaje se justifica debido a que es uno de los lenguajes más ampliamente extendidos y usados actualmente. También debido a que contiene una combinación interesante de retos todavía sin resolver para la depuración algorítmica. Para aumentar todavía más su usabilidad, el depurador ha sido posteriormente adaptado como un plugin de Eclipse, de tal manera que pudiese ser usado por un número más amplio de usuarios. Estos dos depuradores están públicamente disponibles para que cualquier persona interesada pueda acceder a ellos y continuar con la investigación si así lo deseara.

Resum

Hui en dia, els errors no detectats de programació suposen una despesa de milers de milions a l'any per a les empreses i institucions públiques i privades. Tot i això, no hi ha hagut cap avanç significatiu en l'àrea de la depuració que ajude als desenvolupadors durant la fase de desenvolupament del programari. De fet, les mateixes tècniques de depuració que s'utilitzaven fa 20 anys es continuen utilitzant ara. Al llarg del temps, han sorgit algunes alternatives, però encara queda un llarg camí perquè estes siguen prou útils com per a obrir-se camí en el procés de desenvolupament de programari. Una d'elles és la depuració algorítmica, la qual abstrau la informació que el programador ha d'investigar per a depurar el programa, permetent-li d'esta manera centrar-se en el què està ocorrent en compte de en el com. Esta abstracció té un cost: el nivell de granularitat dels errors que poden detectar-se ens permet com a màxim aïllar funcions mal implementades, però no esbrinar quina part d'estes conté l'error. Esta tesi es centra a millorar la depuració algorítmica en molts aspectes. Concretament, els principals objectius d'esta tesi són reduir el temps que l'usuari necessita per a detectar un error de programació així com proporcionar informació més detallada d'on es troba l'error. Per a aconseguir estos objectius, s'han desenvolupat tècniques per a iniciar les sessions de depuració com més prompte millor, reduir el nombre de preguntes que se li formularan a l'usuari, i augmentar el nivell de granularitat dels errors que la depuració algorítmica pot detectar, permetent així continuar buscant l'error inclús dins de les funcions. Com resultat d'esta tesi, s'han definit tres tècniques completament noves, s'ha millorat una tècnica ja existent, i s'han definit dos noves estratègies de depuració algorítmica que milloren les prèviament existents. A més dels resultats teòrics, també s'ha desenvolupat un depurador algorítmic completament funcional que conté i protegix totes estes tècniques i estratègies. Este depurador està escrit en Java i depura codi Java. L'elecció d'este llenguatge es justifica pel fet que és un dels llenguatges més àmpliament estesos i usats actualment. També pel fet que conté una combinació interessant de reptes encara sense resoldre per a la depuració algorítmica. Per a augmentar encara més la seua usabilitat, el depurador ha sigut posteriorment adaptat com un plugin d'Eclipse, de tal manera que poguera ser usat per un nombre més ampli d'usuaris. Estos dos depuradors estan públicament disponibles perquè qualsevol persona interessada puga accedir a ells i continuar amb la investigació si així ho desitjara.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Preamble

1.1 Motivation

Debugging is one of the most important tasks in the software development process. It is necessary in all paradigms and programming languages both during the development and during the maintenance of software systems. In Shapiro's words [92]:

> "*It is evident that a computer can neither construct nor debug a program without being told (...) what problem the program is supposed to solve (...) No matter what language we use to convey this information, we are bound to make mistakes. Not because we are sloppy and undisciplined, as advocates of some program development methodologies may say, but because of a much more fundamental reason: we cannot know, at any finite point in time, all the consequences of our current assumptions.*"

These words describe how arduous programming can become and that no matter how good programmers we are, most probably we will have bugs in our programs. In fact, the debugging process is still one of the most difficult and less automated tasks of software engineering. This is due to the fact that bugs are usually hidden under complex conditions that only happen after particular interactions of software components. Programmers cannot consider all possible computations of their pieces of software, and those unconsidered computations usually produce a bug. In words of Brian Kernighan, the difficulty of debugging is explained as follows:

> *"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"*
>
> The Elements of Programming Style, 2nd edition

The problems caused by bugs are highly expensive. Sometimes more than the product development price. For instance, the NIST report [101] calculated that undetected software bugs produce a cost to the USA economy of $59 billion per year. Even so, the automatization of debugging is still far from being a reality. In fact, during many years, Print Debugging (also known as Echo Debugging) has been the most common method for debugging. Print Debugging allows for easily knowing whether the computation traverses one specific point. Many bugs can be corrected with this information, and the user (maybe optimistically) prefers to use this method before starting a real debugger. Nevertheless, some bugs are almost impossible to detect with Print Debugging, especially in presence of random values, input, and concurrency.

Fortunately, all modern programming environments, e.g., Borland JBuilder [6], NetBeans [76], Eclipse [25], SICStus Prolog SPIDER IDE [93] or ActiveState Komodo [1] include a trace debugger, which allows users to trace computations step by step starting from a point of the execution specified by a breakpoint. In a trace debugger, the user must decide where to place the breakpoints and use them to inspect the values of strategic variables. This process is more difficult in object-oriented programming because the execution of the code is continuously switching between different objects of the system, and thus, it is not obvious for the user where the breakpoints should be placed in the code. Between the point where the effect of a bug is observed, and the point where the bug is located, there can be hundreds or thousands of lines of code, thus, the user must navigate the computation in order to find the bug. Ideally, breakpoints should be placed in a way that the inspected amount of code is reduced as much as possible. Moreover, Trace Debugging is a completely manual task, and the user is in charge of inspecting the computations of the program at a low abstraction level requiring a deep understanding of the source code to find the bug. For this reason, other debugging techniques have been proposed to solve some of these problems, one of them is Algorithmic Debugging.

Algorithmic debugging (AD) [97] is a semi-automatic debugging technique that tries to automate the problem of inspecting a computation in order to find a bug. The idea behind AD is that the own debugger selects automatically the places that should be inspected and asks an oracle (normally the user) about them. The advantage is that the debugger knows a priori the length of the computation, and thus it can perform a dichotomic search by inspecting the subcomputation located in the middle of the remaining suspicious area. This allows the debugger to efficiently explore the computation. Moreover, traditionally, the inspected places are execution of methods and thus, it is easy for the user to decide whether the result produced by an execution with given arguments is actually correct or not. With the user's answers the debugger is able to precisely identify the bug in the source code. Roughly speaking, the debugger discards those parts of the source code associated with correct computations until it isolates a small part of the code (usually a function or procedure) that contains the bug. One interesting property of this technique is that users do not need to see the source code during the debugging sessions, they only need to know the intended and actual results produced by a computation with given inputs. Therefore, if we have available a formal specification of the pieces of software that is able to answer the questions, then the technique is fully automatic.

The most important advantages of AD are its high level of abstraction and its semi-automatic nature thanks to the use of a data structure called Execution Tree (ET). The ET represents the execution of the program where each node represents a concrete execution of a method. The main drawbacks of this technique are:

AD Problem 1: Low scalability. Each ET node needs to record a part of the computation state (the context before and after the execution of the method). Storing the ET of the whole execution may be unpractical.

AD Problem 2: Unnecessary work. The search strategy that traverses the ET may ask unnecessary questions until it reaches the part of the computation that contains the bug.

AD Problem 3: Low granularity. This technique reports a method as buggy, instead of an expression.

AD Problem 4: Concurrency. Algorithmic Debugging has just met concurrency and an implementation and a formal definition is currently being developed to support that feature [15].

1.2 Contributions of the thesis

In this thesis we have solved some of the previously described drawbacks and made a step forward to mitigate the other ones. The techniques developed along this thesis, each of which is useful to solve one of the problems, are the following:

Virtual Execution Trees (Section 5.1) increases the scalability (AD Problem 1) by allowing for storing the ET into a database and starting the debugging session only within a few seconds. This is possible because the debugger can start the debugging session before the ET has been constructed.

Tree Balancing (Section 5.2) improves the ET by introducing new nodes into it that reduce the time needed to debug the program by minimizing the number of questions (AD Problem 2).

Loop Expansion (Section 5.4) uses the *loops2recursion* library (Section 5.3) to expand the nodes of the ET that represent loops, thus allowing for increasing the granularity of the technique, being able to find a bug inside loops (AD Problem 3).

Tree Compression (Section 5.5) removes from the ET the redundant nodes that increase the number of questions needed to find a bug (AD Problem 2).

Optimal Divide & Query (Section 6.2) improves the original version of *Divide & Query* by selecting the optimal node (w.r.t. D&Q strategies) of the ET, and thus reducing the number of questions the user is asked (AD Problem 2).

Divide by Queries (Section 6.4) defines a new search strategy that selects the optimal node (w.r.t. the number of questions the user is asked) for any possible ET (AD Problem 2). We describe the requirements of this strategy, and we provide a naive algorithm that shows that such strategy is possible.

All the proposed theoretical developments are supported by an implementation that was integrated into an algorithmic debugger called Declarative Debugger for Java (DDJ). This debugger was later upgraded to Hybrid Debugger for Java (HDJ). Both of them are open source and publicly available.

DDJ: Declarative Debugger for Java (Section 7.1) An algorithmic debugger that implements all the techniques exposed before: The *Virtual Execution Trees* technique, the *Tree Balancing* technique, the *Loop Expansion* technique, the *Tree Compression* technique, and the *Optimal Divide & Query* strategy.

HDJ: Hybrid Debugger for Java (Section 7.2) A hybrid debugger that is composed of three debugging techniques combined in an Eclipse plugin, one of which is DDJ. The three techniques work together to exploit their strong points while reducing their weakness. The combination of these debugging techniques increases the scalability of DDJ thanks to the use of the standard debugging technique of Eclipse (i.e., its *Trace Debugger*), and the granularity of the errors that are found by the debugger are reduced to expressions thanks to *Omniscient Debugging*.

Both implementations are written in the Java imperative language. Nevertheless, all of the techniques focus on modifying (or generating an improved version of) the ET, thus none of them depends on the language the ET is constructed from. Therefore, despite the examples used along this thesis are written in Java, most of our definitions and algorithms are language- and paradigm-independent.

1.3 Structure of the thesis

The thesis is divided in four main parts: Introduction, Foundations, Algorithmic Debugging, and Conclusions and future work.

1. The Introduction part explains in Section 1.1 the motivations of this thesis, its contributions in Section 1.2, and the explanation of other debugging techniques related with Algorithmic Debugging in Chapter 2.

2. In the Foundations part we explain what Algorithmic Debugging is with explanatory examples, and we show in Chapter 3 a debugging session from the user's perspective. Moreover, all sections of the thesis share the same definitions and notation, which are presented in Chapter 4, to offer a common vocabulary along the whole thesis. However, the thesis has been organized in such a way that each section is self-contained, hence any section can be read without following a concrete order. The explanation of the technique presented in a section may reference some previously developed techniques, but they are briefly explained at that point, if need be. This allows for easily consulting a concrete technique because it is not needed to read and understand other techniques if the reader is interested in only one of them.

3. The Algorithmic Debugging part is divided into four main chapters: techniques, search strategies, implementations, and reformulation.

   - In the techniques chapter (Chapter 5) the five techniques developed during this thesis are explained: *Virtual Execution Trees* in Section 5.1, *Tree Balancing* in Section 5.2, *Loops to recursion* in Section 5.3, *Loop Expansion* in Section 5.4, and *Tree Compression* in Section 5.5. In addition, we also provide a study combining *Loop Expansion* and *Tree Compression* in Section 5.6. At the end of each section, we provide the proofs of the technical results associated with each technique.
   - The search strategies chapter recalls the Divide & Query strategy in Section 6.1 and describes the two new search strategies developed during this thesis: Optimal Divide & Query in Sections 6.2 and 6.3, and Divide by Queries in Section 6.4. Here again, we provide the proofs of the technical results associated with the strategies at the end of the sections.
   - In the implementations chapter we show the two developed debuggers: DDJ in Section 7.1 and HDJ in Section 7.2.
   - Finally, Chapter 8 shows a reformulation of the whole AD technique that, contrarily to the original formulation, is able to represent all the properties and features that the previous techniques need. This reformulation can be also used by other AD researchers to define their own AD, as well as the components that form it (e.g., the ET, the ET nodes, the strategies, the oracle, etc.).

4. With respect to the Conclusions and future work part, it exposes the conclusions of the thesis in Chapter 9, and the open lines of work that can be further explored in Chapter 10.

Finally, in Appendix A we provide a glossary of acronyms that helps the reader to easily and quickly consult and understand a concept when they need it.

Chapter 2

Debugging Techniques

In this chapter we introduce some other debugging techniques related to Algorithmic Debugging. Some of them have been used in some techniques of this thesis, such as Trace Debugging or Omniscient Debugging, whereas other are included due to their close relation to Algorithmic Debugging, such as Abstract Diagnosis, Abstract Debugging, or Delta Debugging.

2.1 Trace Debugging

The most used method for debugging is Trace Debugging (TrD). It allows the user to traverse the trace of a computation step by step. The user places a *breakpoint* in a line of the source code and the debugger stops the computation when this line is reached. Then, the user proceeds line by line and, at each step, the user can inspect the state of the computation (i.e., value of variables, exceptions, etc.). During the traversal of the trace, when a call to a method is reached, the user can decide either to enter into the method (*step into*) or to skip it (*step over*). Modern breakpoints are conditional, i.e., the breakpoint includes conditions about the values of some variables, or about the performed action where they are defined. For instance, it is possible to define a breakpoint that only stops the computation when an exception happens, or when a specific class is loaded.

Trace Debugging has one important advantage over other debugging techniques: scalability. The debugger only needs to take control over the interpreter to normally execute the program. Hence, its scalability is the same as the one of the interpreter. On the other hand, Trace Debugging has four main drawbacks:

TrD Problem 1. The whole debugging process is done at a very low abstraction level. The users just follow the steps of the interpreter, and they need to understand how the value of variables change to identify an error.

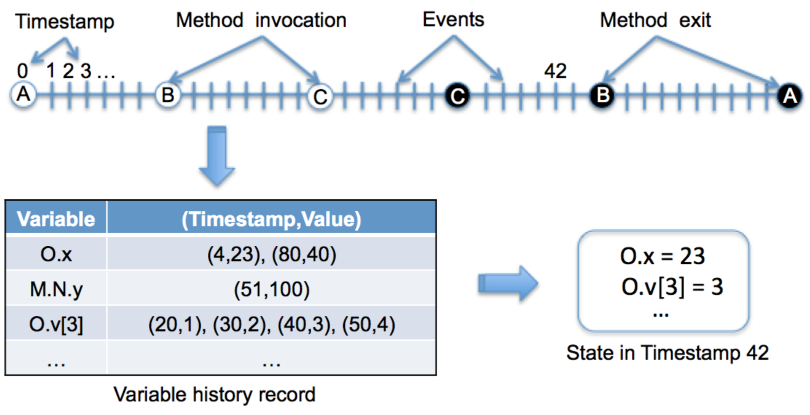
TrD Problem 2. The debugger can generate an overwhelming amount of information.

TrD Problem 3. The debugging process is completely manual. The users use their intuition to place the breakpoints. If the breakpoint is after the bug, they have to place it before, and restart the program. If the breakpoint is placed long before the bug, then they have to manually inspect a big part of the computation.

TrD Problem 4. The inspection of the computation is made forwards, while the natural way of discovering the bug is backwards from the bug symptom.

2.2 Omniscient Debugging

Omniscient debugging [60] (OD) solves the fourth problem of Trace Debugging (TrD Problem 4) with the cost of sacrificing scalability. Basically, both techniques rely on the use of breakpoints and they both do exactly the same from a functional point of view. The difference is that Omniscient Debugging allows the user to trace the computation forwards and backwards (chronologically). This is very useful because it allows the user to perform steps backwards from the bug symptom. To do this, the debugger needs a mechanism to reconstruct every state of the computation. One of the most scalable schemas to do this is depicted in Figure 2.1. In this figure, we have an horizontal line representing an execution as a sequence of events. Some of these events are method invocations (represented with a white circle), and method exits (represented with a black circle). Each event is identified with a timestamp. From the execution, the omniscient debugger stores a variable history record that contains the values of all variables together with the exact timestamp where each value was updated. The omniscient debugger also stores information about the scope of variables that we omit here for clarity. With this information the debugger can reconstruct any state of the computation. For instance, in state 42, value `M.N.y` did not exist, and the last value of variables `O.x` and `O.v[3]` were 23 and 3, respectively.



Variable	(Timestamp,Value)
O.x	(4,23), (80,40)
M.N.y	(51,100)
O.v[3]	(20,1), (30,2), (40,3), (50,4)
...	...

Figure 2.1: Timestamps-based scheme to store traces in OD

Being able to reconstruct the complete trace also allows the user to start the execution at any point. Nevertheless, storing all values that each variable in an execution has taken is usually impossible for realistic industrial (large) programs, and even for medium sized programs. Thus scalability is very limited in this technique.

2.3 Abstract Diagnosis

Abstract Diagnosis [21] is a debugging technique that automatically searches and finds bugs in a program. First of all the programmer has to write the program as well as an intended semantics (a.k.a. specification) that defines how the program should work. Then, the code written by the programmer is given to the technique and, by selecting which part of the code the programmer is interested in, it automatically uses the abstract interpretation [22] technique to obtain the properties that this part of the code holds. Finally, these properties are compared against the intended semantics to check whether the properties violate it. If it does, an error is found in this part of the code.

This technique is very similar to Algorithmic Debugging. The main difference is that in Algorithmic Debugging the oracle (normally the user) is in charge of comparing the results of a computation with what they expected. On the contrary, Abstract Diagnosis needs a specification (normally written by the user) and the own tool is in charge of obtaining the properties the program has and checking them against the specification.

2.4 Abstract Debugging

Abstract Debugging [7] is a debugging technique based on the Abstract Interpretation technique [22]. Abstract Interpretation infers the possible values that the variables of a program can obtain at any point of an execution. Abstract Debugging uses this information to automatically debug a program. First of all a set of assertions has to be inserted into the source code. These assertions are normally introduced by the user and are considered to be conditions that some variables have to fulfil at the point where the assertion is introduced, otherwise the program does not work as expected by the programmer and an error is manifested. Two possible types of assertions exist depending on how many times the conditions have to be fulfilled: *invariant assertion*, which are conditions that have to be always held; and *intermittent assertions*, which are conditions that have to be eventually held. Hence, the debugging technique only has to use abstract interpretation to determine whether these conditions are fulfilled in all possible executions of the program. Let us see how the debugging technique works in the following example:

Example 2.4.1 *Consider the following Java code:*

```
  (1) int[] numbers = { 1, 2, 3, 4, 5 };
  (2) for (int i = 0; i < 5; i++)
  (3) {
* // assert(0 <= i < numbers.length)
  (4)     int number = numbers[i];
  (5)     System.out.println(number);
  (6) }
```

*Here the user has introduced an assertion (showed in the code with *) indicating that* $0 \leq i <$ *numbers.length to ensure that the program cannot throw an IndexOutOfBoundException exception at run-time. Imagine that some time in the future a developer decides to replace line (1) with:*

```
int[] numbers = { 1, 2, 3, 4 }
```

At that point, the debugger would automatically detect that the invariant is violated due to the possibility that the i variable contains the value 4, which would violate the assertion $0 \leq i < numbers.length$ *because numbers.length would value* 4*.*

2.5 Delta Debugging

Delta Debugging [105] is a debugging technique used to detect the part of the program that makes it crash. This is done by inspecting the differences (the deltas) from one correct program against its new release that does not work properly. If these deltas can be automatically detected, then this technique automatically finds which is the single or combination of deltas that produce the bug. The idea behind this theory is that if we have a code with an error and a set of deltas added after the last correct code, we can use delta debugging to incrementally remove those deltas from the buggy code until the code works again properly, thus detecting which deltas produce the bug. These deltas are not removed by the technique one by one, but using combinations of deltas that are called configurations. The algorithm of the technique assumes an initial working configuration composed of all the deltas, and this working configuration is modified as soon as a subconfiguration of the working configuration is proved to not produce the bug. The algorithm finishes when no subconfiguration can be removed. Let us show the technique in the following example:

Example 2.5.1 *Consider the next HTML code in which each delta (a new line in the code) is associated with an identifier on the left. Note that we have used lines as deltas, another option would be to use elements or even characters.*

```
<html>
  <head>
d1) <title>David Insa Cabrera</title>
d2) <base href="http://www.dsic.upv.es/~dinsa/">
  </head>
  <body>
    <div id="global">
      <div id="header">
d3)     <img id="globe" alt="" src="resources/images/misc/World.png">
d4)     <div id="author">David Insa Cabrera</div>
      </div>
      <div id="navigation">
d5)     <div id="icon"><ing src="resources/icons/menu/home.png" alt=""></div>
        <ul>
          <li class="home selected"><a href="en/?section=home">Home</a>
          <li class="research"><a href="en/?section=research">Research</a>
          <li class="publications"><a href="en/?section=publications">Publications</a>
          <li class="visited_places"><a href="en/?section=visited_places">Visited places</a>
d6)       <li class="contact"><a href="en/?section=contact">Contact</a>
        </ul>
      </div>
      <div id="footer">Last update: November 04, 2015</div>
    </div>
  </body>
</html>
```

The following table would correspond to a delta debugging session where each `dx` *is associated with the delta of the same identifier in the HTML code. Each* `dx` *in a cell of the delta columns means that the associated line is kept in the HTML code whereas* $-$ *means that this delta has been discarded of having the blame. Finally, a* $\times$ *in the result column means that the bug still happens using this configuration whereas* ✓ *means that this configuration does not produce the bug. Note that the configuration is marked with* $\times$ *when the same bug is found, in fact when a different bug is found this configuration is also marked with* ✓.[1]

Test	*Configuration*						*Result*
	d1	*d2*	*d3*	*d4*	*d5*	*d6*	
Test 1	*d1*	*d2*	*d3*	*d4*	*d5*	*d6*	$\times$
Test 2	*d1*	*d2*	*d3*				✓
Test 3				*d4*	*d5*	*d6*	$\times$
Test 4	–	–	–	*d4*	*d5*		$\times$
Test 5	–	–	–	*d4*		–	✓
Test 6	–	–	–		*d5*	–	$\times$
Bug	–	–	–	–	*d5*	–	

In the table we can see that after checking there exists a bug using the initial configuration (Test 1), delta debugging uses a subconfiguration to try to check whether the bug is produced by one of the deltas of that subconfiguration. These subconfigurations are initially composed of the first half of the working configuration (Test 2) and then, after checking that the bug does not happen, by the second half (Test 3). Note that each time the configuration of the test has produced a $\times$ *result, the deltas that do not belong to that configuration are discarded, so they cannot have the blame of producing the bug. Hence, after checking that Test 3 produces the bug, only all its deltas are considered again in the rest of the process. Here again, the first half of the working configuration is checked (Test 4) and, because it produces the*

[1]When the current configuration produces a code that cannot be used to check whether the bug still happens (e.g., the code cannot be compiled) the result of the test is **?** and it is treated as ✓.

*bug, d*6 *is discarded from having the blame. The same process is repeated in Test 5, but in this case the test passed. Hence the second half of the working configuration is checked (Test 6). Finally, delta debugging finds the bug in the deltas of the configuration of Test 6 (i.e., the bug is produced when d*5 *is in the HTML code). Note that the bug is produced because the* `img` *element has been wrongly typed as* `ing`.

Part II

Foundations

Chapter 3

Algorithmic Debugging from the User's Perpective

Algorithmic debugging (AD) [92] is a semi-automatic debugging technique that has been extended to practically all paradigms [95, 97]. The technique is based on the external oracle's answers (typically the user) to a series of questions automatically generated by the algorithmic debugger. The questions are always whether the result of the execution of a method with given input values is actually correct. Therefore, the user only needs to know what a function is supposed to do (instead of how) in order to debug it. The user's answers provide the debugger with information about the correctness of some (sub)computations of a given program; and the debugger uses them to guide the search for the bug until a buggy portion of code is isolated.

Example 3.0.1 *Consider the Java program in Figure 3.1 that simulates Tic-Tac-Toe games. The* Replay *class reads from a file a new game and it reproduces the game using a* TicTacToe *object. This program is buggy (it contains two bugs), and thus it does not produce the expected marks on the board.*

Each AD session can identify one single bug, therefore we need to perform two sessions to find both bugs. A first AD session for this program is shown below, where boards are represented with a picture for clarity (e.g., considering that the null character in Java is `'\u0000'` *and that we represent it with* `''`*, then* $\{\{O,'','' \}\{X,'',''\}\{'','',''\}\}$ *is represented with). For the time being ignore the* **`Node`** *column:*

```
Starting Debugging Session...
 Node      Initial context                 Method call              Final context          Answer
 (2)    [turn='X',board=     ]    game.mark('X',0,0)        [turn='O',board=     ]  ?  YES

 (7)    [turn='O',board=     ]    game.mark('O',0,1)        [turn='', board=     ]  ?   NO

 (8)    [turn='X',board=     ]    game.win(0,1)=true        [turn='X',board=     ]  ?   NO

 (9)                              equals('X','','')=false                           ?  YES

 (10)                             equals('','','')=true                             ?  YES

Bug located in method:  win(int row, int col) of the TicTacToe class
```

Note that the debugger generates questions, and the user only has to answer the questions with `YES` *or* `NO`*. It is not even necessary to see the source code. Each question is about an execution of a particular method, and the user answers* `YES` *if the execution is correct (i.e., the output and the final context are correct) and* `NO` *otherwise.*

```
public class Replay {
    public static void main(String[] args) throws IOException {
        TicTacToe game = new TicTacToe();
        FileReader file = new FileReader("./game.rec");
        play(game, file);
    }
    private static void play(TicTacToe game, FileReader file) throws IOException {
        BufferedReader br = new BufferedReader(file);
        String line = br.readLine();
        while ((line = br.readLine()) != null) {
            char player = linea.charAt(0);
            int row = Integer.parseInt(line.charAt(2) + "");
            int col = Integer.parseInt(line.charAt(4) + "");
            game.mark(player, row, col);
        }
    }
}

public class TicTacToe {
    private static boolean equals(char c1, char c2, char c3) {
        return c1 == c2 && c2 == c3;
    }

    private char turn = 'X';
    private char[][] board = new char[3][3];

    public void mark(char player, int row, int col) {
        if (turn == '\u0000' || turn != player
            || row < 0 || row > 2 || col < 0 || row > 2
            || board[row][col] != '\u0000')
            return;
        board[col][row] = player; // Bug!! Wrong position
        // Correct: board[row][col] = player;
        turn = turn == 'X' ? 'O' : 'X';
        if (win(row, col))
            turn = '\u0000';
    }
    private boolean win(int row, int col) {
        // Bug!! Didn't check whether the input position is empty
        // Correct: if (board[row][col] == '\u0000') return false;
        if (equals(board[row][0], board[row][1], board[row][2]))
            return true;
        if (equals(board[0][col], board[1][col], board[2][col]))
            return true;
        if (col == row && equals(board[0][0], board[1][1], board[2][2]))
            return true;
        if (col + row == 2 && equals(board[0][2], board[1][1], board[2][0]))
            return true;
        return false;
    }
}
```

Figure 3.1: Tic-Tac-Toe Java program

At the end, the debugger points out the part of the code that contains the bug. In this case, the TicTacToe.win *method is wrong. This method checks whether three adjacent positions of the board are equal and if so, it returns* `true`*. This is correct if the three positions contain either* `'X'` *or* `'O'`*; but it is wrong if the three positions are empty. This error can be easily corrected by checking that the value of the input position is not null (i.e., adding at the beginning of the* TicTacToe.win *method the following sentence:* `if (board[row][col] == '\u0000') return false;`*).*

Traditionally, AD is a two-phase process: (1) The construction of a data structure —the so-called *Execution Tree* (ET) [78]— representing the execution of the program including all subcomputations; and (2) the exploration of the ET with a given search strategy to ask questions and process the user's answers to locate the bug.

In the object-oriented paradigm the ET is constructed as follows: The root node is (usually) the *main* function of the program; for each node n with associated method m, and for each execution of a method triggered from the definition of m, a new node is recursively added to the ET as a child of n. Each node of the ET contains an equation that consists of an execution of a method with completely evaluated arguments and results. The node also contains additional information about the context of the method before and after its execution (values of attributes and global variables in the scope of the method).

Example 3.0.2 *Consider again the Java program in Figure 3.1. Figure 3.2 depicts the portion of the ET associated with the execution of the* `play(game, file)` *method using* game.rec *as the input file. Each node contains:*

- *A string representing the method call (including input and output) depicted at the top of each node.*
- *The variables (and their values) in the scope at the beginning and at the end of the execution of the method. When the value of a variable is modified during the execution of the method, the node contains both values on the left and on the right of the node, respectively. When the variable is not modified, it is shown only once in the middle of the node.*

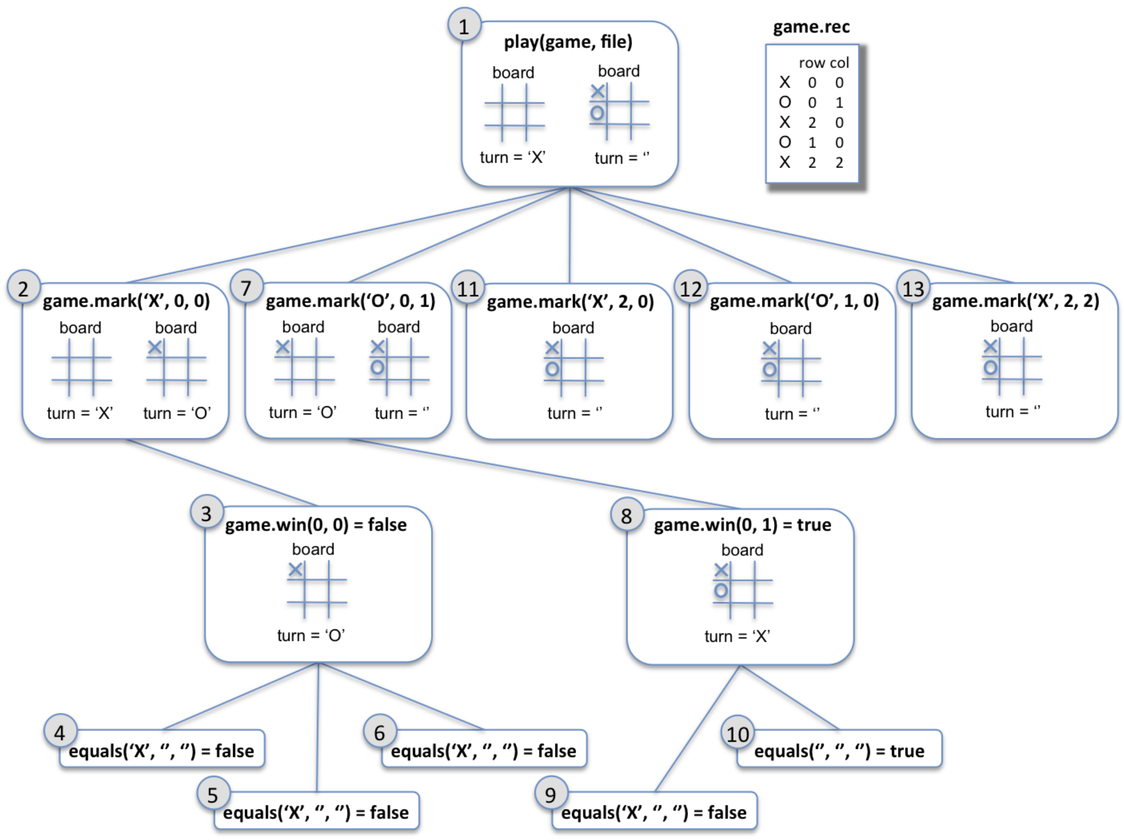


Figure 3.2: ET associated with the `play(game, file)` call of the program in Figure 3.1

Once the ET is built, in the second phase, the technique produces a dialogue between the debugger and the user to find bugs. This dialogue corresponds to questions about the information stored inside the nodes that the user has to answer with either YES or NO. For instance, each question in the debugging session of Example 3.0.1 corresponds to a node (see the `Node` column) of the ET in Figure 3.2. These nodes have been selected using the Divide & Query strategy [92]. Essentially, the answers rely on the programmer having an *intended interpretation* of the program. In other words, some computations of the program are correct and others are wrong with respect to the programmer's intended semantics. Therefore, algorithmic debuggers compare the results of subcomputations with what the programmer intended. After every answer, the algorithmic debugger uses this comparison to mark some nodes of the ET as correct or wrong. When all the children (if any) of a wrong node are correct, the node becomes buggy and the debugger locates the bug in the part of the program associated with this node [64]. Hence, by asking the user questions or using a formal specification, the system can precisely identify the location of a bug.

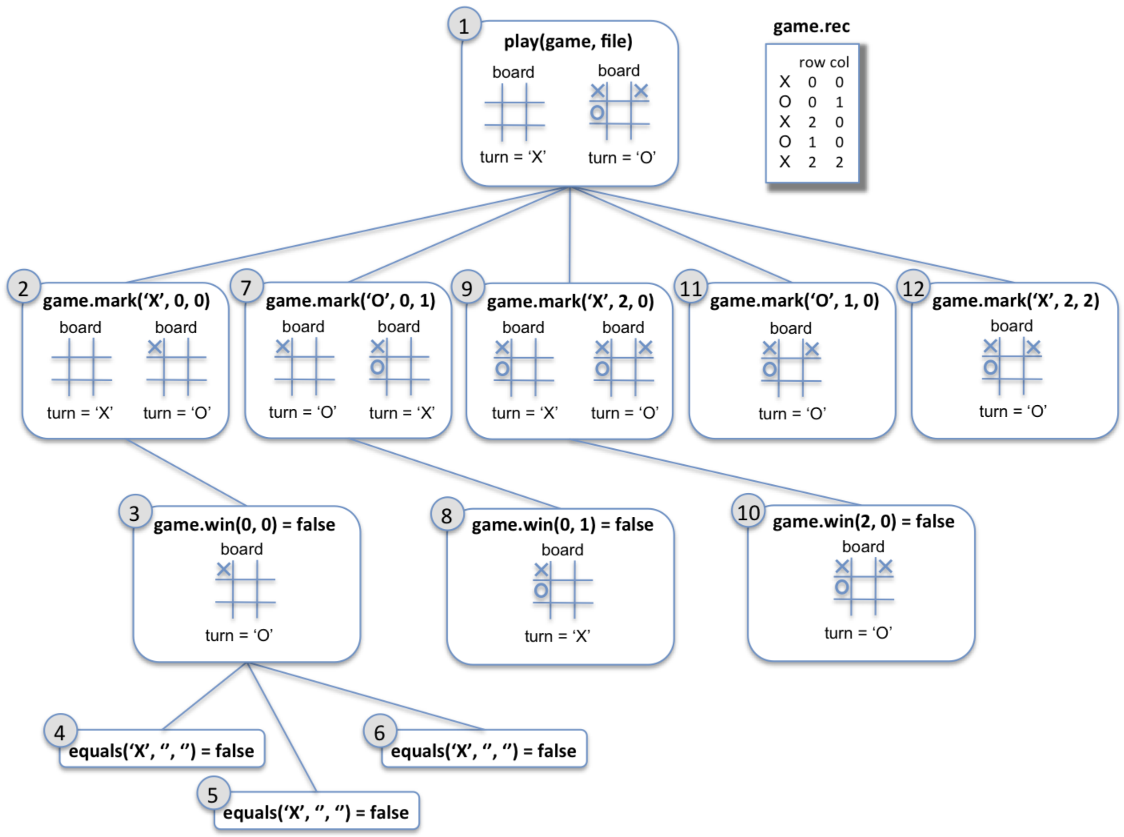


Figure 3.3: ET associated with the `play(game, file)` call of the partially corrected version of the program in Figure 3.1

Example 3.0.3 *After correcting the bug located in Example 3.0.1, the program still produces an incorrect result. In Figure 3.3 we see the produced ET obtained by executing the partially corrected code. The AD session using the partially corrected program is the following:*

```
Starting Debugging Session...
 Node     Initial context         Method call              Final context       Answer
  (2)   [turn='X',board=    ]   game.mark('X',0,0)       [turn='O',board=    ] ?  YES

  (7)   [turn='O',board=    ]   game.mark('O',0,1)       [turn='X',board=    ] ?   NO

  (8)   [turn='X',board=    ]   game.win(0,1)=false      [turn='X',board=    ] ?  YES

Bug located in method:  mark(char player, int row, int col) of the TicTacToe class
```

The debugger points out the part of the code that contains the bug. In this case, `board[col][row] = player` *should be* `board[row][col] = player`*.*

Chapter 4

Preliminary Definitions and Notation

Along this thesis, the term *debugging* is used to refer to the phase in the software development process in charge of finding and correcting bugs. However, Algorithmic Debugging (AD), despite what its name implies, does not actually debug bugs, but it just detects them. In particular, AD defines how to find bugs, but the action of correcting them corresponds ultimately to the users. Hence, we classify AD as a debugging technique whose goal is to detect bugs. We also need to clearly state what a bug is, and thus, in the rest of the thesis, a bug is considered as *any portion of code that does not produce the programmer's expected behaviour.* This definition includes:

1. Those codes whose execution produces a wrong result. In AD, this means that the right-hand-side of the generated question is wrong.

2. Those codes whose execution produces an incomplete result. In AD, this means that some answers or results are missing on the right-hand-side of the generated question.

3. Those codes whose execution is inefficient (compared with what the programmer intended). AD cannot detect this kind of bugs.

Note that, according to the definition, a code that is fundamentally slow does not necessarily contain a bug, because there exist algorithms that cannot be executed in an efficient way or in a practical time. This is the case for instance of the algorithms that calculate the π number. It does not matter how the algorithm is implemented, the time that is needed to obtain the π number is beyond the current state of the art (mainly because it is supposed to be a number with infinite decimals). Hence, those algorithms that calculate the π number cannot be classified as buggy just because they do not obtain it in a practical time. Only those algorithms that do not perform what it is expected from them should be considered as buggy.

Because bugs of type 3 are not detectable by AD, they are not considered in the techniques proposed in this thesis, which only try to detect bugs of types 1 and 2. For further information about bugs of type 2 (missing results), which are more related with logic programming and non-deterministic systems, we refer the interested reader to specific works such as [89, 11, 87]), which will provide more detailed explanations and examples about how to treat them.

In the rest of this chapter we provide the definitions and notations of AD used along the whole thesis. Specifically, we provide a general and complete definition of Nodes and Execution Trees (ET), and we show and explain the most used search strategies of AD in the literature. More concrete definitions of each component may be provided at the beginning of each technique proposed along the thesis, if need be. For the sake of concreteness, we use along the whole thesis the Java language as the basis of the definition of ETs and to show examples of AD. AD was introduced in the logic paradigm and later extended to the functional and imperative paradigm via a sightly adaptation of the ET, but, once the ET is constructed, the technique behaves in the same way for all paradigms.

4.1 Execution Tree Nodes

In this section we show how the information of the execution of the program is stored in the nodes of a tree that represents that computation. To do this, we previously need to formally define the notion of context.

Definition 4.1.1 (Context) *Let $\mathcal{P}$ be a program, and X the execution of a method in $\mathcal{P}$. The* context *of X at a particular instant t is $\{(a, v) \mid a$ is a variable in the scope of X at instant t and v is the value of $a\}$.*

Roughly, the context of a method at a particular instant of its execution is composed of all the variables of the program that are visible at this point. Clearly, a variable of a context can be an object that in turn contains other variables. In a realistic program, each node contains several data structures that could change during the execution. The context at the beginning and at the end of the execution of the method should be visualized together with the call to the method, so that the user can decide whether the execution of the method computes the expected result.

Definition 4.1.2 (Method Execution) *Let $\mathcal{P}$ be a program and X an execution of $\mathcal{P}$. Then, we call to each execution of a method performed along X as* method execution *and it is represented with a triple $\mathcal{E} = (b, m, e)$ where m represents the call to the method with its arguments and the returned value, b is the context of the method in m at the beginning of its execution, and e is the context of the method in m at the end of its execution.*

Note that both contexts, b and e, include those variables in the scope of the method but not used along its execution; because the bug could be produced by not updating a variable that should be updated. Method executions are the questions asked by the debugger. For instance, a method execution $\mathcal{E} = (b, m, e)$ can correspond to a question along the lines of: *Should the execution of m within the b context produce the e context?*, or *m produced e from b, is that correct?*. The user's answer can be either YES (to denote that this computation is correct) or NO (to denote that the computation is wrong).[1]

4.2 Execution Tree, Marked Execution Tree

Thanks to the declarative properties of AD, we can ignore the operational details of an execution. From the point of view of the debugger, an execution is a finite tree where each node represents a method execution. This can be modeled with the following grammar:

$$T = (b, m[L], e) \qquad L = \epsilon \qquad L = TL$$

where the terminal m is a method of the program and b and e represent the context at the beginning and at the end of the execution of the method.

Roughly speaking, an ET is a tree whose nodes represent method executions and the parent-child relation is defined by the tree produced by the grammar. Formally,

[1]For the sake of concreteness, we ignore other possible answers such as "I don't know" or "I trust this function". They are accepted in our implementation, but we refer the interested reader to [95] for theoretical implications of their use.

Definition 4.2.1 (Execution Tree) *Given a program $\mathcal{P}$ and a method execution $\mathcal{E}$, the* execution tree *(ET) of $\mathcal{P}$ with respect to $\mathcal{E}$ is a tree $T = (N, E)$ where $\forall n \in N$, n is a method execution, and*

- *The root of the ET is $\mathcal{E}$.*
- *For each pair of method executions $\mathcal{E}_1$, $\mathcal{E}_2 \in N$, we have that $(\mathcal{E}_1 \rightarrow \mathcal{E}_2) \in E$ iff*
  1. *from the instant when $\mathcal{E}_1$ starts until the instant it ends, $\mathcal{E}_2$ starts, and*
  2. *from the instant when $\mathcal{E}_1$ starts until the instant when $\mathcal{E}_2$ starts, there does not exist a method execution $\mathcal{E}_3$ such that $\mathcal{E}_3$ has started but not ended.*

Note that we use $(n \rightarrow n')$ to denote a directed edge from n to n'. In the following we use E^+ to refer to the transitive closure of E, and E^* for the reflexive and transitive closure. Moreover, we say that the weight of a node is the number of nodes contained in the tree rooted at this node. We refer to the weight of a node n as w_n.

Because the ET is modified along the debugging session due to the user's answers, we also define what a *marked execution tree* is. Basically, it is an ET where some nodes have been marked as correct (i.e., answered YES), some nodes have been marked as wrong (i.e., answered NO), and the correctness of the other nodes is undefined.

Definition 4.2.2 (Marked Execution Tree) *A* marked execution tree *(MET) is a tree $T = (N, E, M)$ where N are the nodes, $E \subseteq N \times N$ are the edges, and $M : N \rightarrow V$ is a marking total function that assigns to each node in N a value in the domain $V = \{Correct, Wrong, Undefined\}$.*

Along the thesis we will use either ETs or METs depending on whether or not the marking of the tree is relevant for the proposed technique.

On the practical side, some AD implementations do not keep the whole tree, but they remove from the tree those nodes than cannot contain the bug. In these implementations, all nodes in the MET are initially marked as *Undefined*. But with every user's answer, a new MET is produced. Concretely, given a MET $T = (N, E, M)$ and a node $n \in N$, the user's answer to the question in n produces a new MET similar to the previous one where:

- if the answer is YES, then this node and its subtree are removed from the new MET.
- If the answer is NO, then all the nodes in the new MET are removed except this node and its descendants.

Note that, in the new recalculated MET the root is the only node that can be marked as *Wrong*. Moreover, the rest of nodes can only be marked as *Undefined* because when the answer is YES, the associated subtree is deleted from the MET.

Despite algorithmic debuggers normally implement the recalculated MET version, both of them represent and provide the same information, so any of them can be used without loss of precision. In fact, in our implementation we use the MET of Definition 4.2.2 allowing for undoing modifications made in the tree, but along this thesis we use the recalculated MET version due to its intuitiveness and simplicity while proving theorems. From here on, when we refer to the MET, we are actually referring to the recalculated version.

Note that the size of the MET is gradually reduced with each answer. If we delete all nodes in the MET, then the debugger concludes that no bug has been found. If, contrarily, we finish with a MET composed of a single node marked as wrong, this node is called the *buggy node* and it is identified as the one responsible for a bug in the program. All this process is defined in Algorithm 1 where the *selectNode* function selects a node in the MET the user will be asked about using the *askNode* function.

Algorithm 1 General algorithm for AD

```
Input: A MET $T = (N, E, M)$
Output: A buggy node or $\perp$ if no buggy node is detected
Preconditions: $\forall n \in N, M(n) = Undefined$
Initialization: $buggyNode = \perp$

begin
  1) while $(\exists n \in N, M(n) = Undefined)$
  2)     $node = selectNode(T)$
  3)     $answer = askNode(node)$
  4)     if $(answer = Wrong)$ then
  5)         $M(node) = Wrong$
  6)         $buggyNode = node$
  7)         $N = \{n \in N \mid (node \rightarrow n) \in E^*\}$
  8)     else
  9)         $N = N \backslash \{n \in N \mid (node \rightarrow n) \in E^*\}$
 10)     end if
 11)     $E = \{(n \rightarrow n') \in E \mid n, n' \in N\}$
 12) end while
 13) return $buggyNode$
end
```

Algorithm 1 keeps asking the user questions until a buggy node is found. This node is easily detected because Algorithm 1 only has to find in the initial ET a node that computes a wrong result whereas its children (if any) do not.

Definition 4.2.3 (Buggy node) *Given a MET $T = (N, E, M)$, a buggy node of T is a node $n \in N$ such that:*

- *$M(n) = Wrong$, and*
- *$\forall n' \in N$, $(n \rightarrow n') \in E$, $M(n') = Correct$.*

Note that Definition 4.2.3 stands for both MET versions we have provided. In the version of Definition 4.2.2 the children of a buggy node are kept in the MET and are marked as *Correct*, whereas in the recalculated MET these children have been removed from the MET (so the second condition of Definition 4.2.3 is also satisfied).

According to Definition 4.2.3, when all the children of a node with a wrong computation (if any) are correct, the node becomes buggy and the debugger locates a bug in the part of the program associated with this node [64]. A buggy node detects a *buggy method*, which informally stands for methods that may compute an incorrect context even though all the method executions performed by them are correct.

This finally takes us to the main properties of AD: its correctness and completeness. These properties show that, when a bug symptom is detected, AD is useful to find a bug.

Theorem 4.2.4 (Correctness of AD [64]) *Given an ET with a buggy node n, the method associated with n contains a bug.*

Theorem 4.2.5 (Completeness of AD [92]) *Given an ET with a bug symptom (i.e., the root is associated with a method with a wrong final context), provided that all the questions generated by the debugger are correctly answered, then, a bug will eventually be found.*

4.3 Algorithmic Debugging search strategies

AD search strategies are functions that select from a MET which is the next node the user will be asked about. There exist a lot of AD search strategies in the literature, but almost all of them can be classified into two main groups: Top-Down and Divide & Query. In this section we show the four most used search strategies nowadays and we refer the interested reader to [97] to see more variants of Top-Down search [69, 24], variants of Divide & Query search [44], some others [68, 97], and a comparison of all search strategies defined so far.

4.3.1 Top-Down strategies

Due to the fact that questions are asked in a logical order (i.e., consecutive questions refer to related parts of the computation), *Top-Down* is the search strategy that has been traditionally used (see, e.g., [10, 12, 58]) to measure the performance of different debugging tools and methods. The first version of Top-Down was created by Shapiro [92] and it is called *Top-Down Left-to-right*. It basically consists of a top-down (assuming that the root is at the top) left-to-right traversal of the ET. When the answer to the question of a node is NO, then the next question is associated with the leftmost non-marked node of its children. When the answer is YES, the next question is associated with the leftmost non-marked node of its siblings. Therefore, the asked node is always a child or a sibling of the previously asked node. Hence, the idea is to follow the path of wrong computations from the root of the tree to the buggy node.

However, selecting always the leftmost child does not take into account the size of the subtrees that can be explored. Binks proposed in [5] a new Top-Down search in order to consider this information when selecting a node. This variant is called *Heaviest First* because it always selects the child or sibling (depending on the answer) with the biggest subtree. The objective is to avoid selecting small subtrees that have a lower probability of containing a bug.

4.3.2 Divide & Query strategies

Another important search strategy is *Divide & Query* (D&Q) [92], which always selects the node whose size of its subtree is the closest to half the size of the whole tree. If the answer is YES, this node (and its subtree) is pruned. If the answer is NO, the search continues only in the subtree rooted at this node. This search strategy asks, in general, fewer questions than Top-Down searches because it prunes near the half of the tree with every question. Therefore, D&Q has a $O(n * log\ n)$ cost with respect to the number of questions being n the number of nodes of the ET. However, it has the drawback that the selected questions depend on the size of the tree instead of depending on the previous questions, and thus the questions may not be highly related to the previous ones causing the user to lose the general view of the computation being debugged.

The main objective of D&Q is to simulate a dichotomic search by selecting the node that better divides the MET into two subMETs with a weight as similar as possible. Therefore, given a MET with n remaining nodes, D&Q searches for the node whose weight is closer to $\frac{n}{2}$. The original algorithm by Shapiro [92] always selects:

- the heaviest node n' whose weight is as close as possible to $\frac{n}{2}$ with $w_{n'} \leq \frac{n}{2}$

Hirunkitti and Hogger noted that this is not enough to divide the MET in half and their improved version [44] always selects the node whose weight is closer to $\frac{n}{2}$ between:

- the heaviest node n' whose weight is as close as possible to $\frac{n}{2}$ with $w_{n'} \leq \frac{n}{2}$, or
- the lightest node n' whose weight is as close as possible to $\frac{n}{2}$ with $w_{n'} \geq \frac{n}{2}$

Part III

Algorithmic Debugging

Chapter 5

Algorithmic Debugging Techniques

In this chapter we introduce the four techniques developed along this thesis, namely: Virtual Execution Trees, Balancing Trees, Loop Expansion, and Tree Compression. The Loop Expansion technique requires the *loops2recursion* library that has also been implemented from scratch and explained in Section 5.3. In addition, Loop Expansion and Tree Compression are two independent techniques that can be easily combined to obtain better results. Therefore, in Section 5.6 we evaluate the improvement obtained by using both techniques at the same time.

5.1 Virtual Execution Trees

5.1.1 Introduction

Algorithmic Debugging (AD) is based on a data structure called Execution Tree (ET) (see Chapter 4 for the conventional definition and explanation of ETs, and see Chapter 8 for a redefinition and generalization of ETs). Unfortunately, with realistic programs, the ET may be huge (indeed in gigabytes) and this is the main drawback of this debugging technique, because the lack of scalability has not been solved yet: If ETs are stored in main memory, the debugger is out of memory with big ETs that do not fit in. If, on the other hand, they are stored in a database, debugging becomes a slow task because some questions need to explore a big part of the ET; and also because storing the ET in the database is a time-consuming task. Some advances have been done to improve the scalability of the technique. For instance, in [28] the authors improved the technique by using HOOD to obtain the ET, which avoids them from instrumenting the whole source code. But it does not matter how much the creation of the ET is improved, it still has to be stored somewhere.

In some languages, the scalability problem is inherent to the current technology that supports the language and cannot be avoided with more accurate implementations. For instance, in Java, current algorithmic debuggers (e.g., JavaDD [36] and DDJ [12]) are based on the *Java Platform Debugger Architecture* (JPDA) [72] to generate the ET. This architecture uses the *Java Virtual Machine Tools Interface*, a native interface that helps to inspect the state and to control the execution of applications running in the *Java Virtual Machine* (JVM). Unfortunately, the time scalability problem described before is also present in this architecture, and hence, any algorithmic debugger implemented using the JPDA will suffer from scalability problems. For instance, we conducted some experiments to measure the time needed by JPDA to produce the ET[1] of a collection of medium/large benchmarks (e.g., an

[1]These times correspond to the execution of the program, the production of the ET and its storage in a database.

interpreter, a parser, a debugger, etc). Results are shown in the `ET time` column of Table 5.1 (the rest of the columns can be ignored for the time being).

`Benchmark`	`var. num.`	`ET size`	`ET time`	`node time`	`cache lim.`	`ET depth`
`argparser`	8.812	2 MB	22 s.	407 ms.	7	7
`cglib`	216.931	200 MB	230 s.	719 ms.	14	18
`kxml2`	194.879	85 MB	1318 s.	1844 ms.	6	9
`javassist`	650.314	459 MB	556 s.	844 ms.	7	16
`jtstcase`	1.859.043	893 MB	1913 s.	1531 ms.	26	57
`HTMLcleaner`	3.575.513	2909 MB	4828 s.	609 ms.	4	17

Table 5.1: Benchmark execution results

Note that, in order to generate the ET, the JVM in combination with JPDA needs some minutes, thus the debugging session would not be able to start until this process finishes.

In this section we propose a new implementation model [47] that solves the two scalability problems, namely, memory and time. Because it is not always possible (e.g., in Java) to quickly generate the ET, the process of generating the ET may cause a bottleneck in AD. Therefore, our model is based on the following question: *Is it possible to start the debugging session before having computed the whole ET?* The answer is yes.

We propose a framework in which the debugger uses the (incomplete) ET while it is being dynamically generated. Roughly speaking, two processes run in parallel. The first process generates the ET and stores it into both a database (the whole ET) and main memory (a part of the ET). The other process starts the debugging session only using the already generated part of the ET. Moreover, we use a three-cache memory system to speed up the debugging session and to guarantee that the debugger is never out of memory.

The rest of the section has been structured as follows: In Section 5.1.2 we introduce a new implementation architecture for algorithmic debuggers and discuss how it solves the two scalability problems shown before. In Section 5.1.3 we give some details about the implementation and show the results obtained with a collection of benchmarks.

5.1.2 A new architecture for Algorithmic Debugging

This section presents a new architecture in which AD is not done in two sequential phases, but in two concurrent phases; that is, while the ET is being generated, the debugger is able to produce questions. This new architecture solves the scalability problems of AD. In particular, we use a database to store the whole ET, and only a part of it is loaded into main memory.

Moreover, in order to make the algorithms that traverse the ET independent of the database caching problems, we use a three-tier architecture where all the components have access to a *Virtual Execution Tree* (VET). The VET is a data structure which is identical to the ET except that some nodes are missing (not generated yet) or incomplete (they only store a part of the method execution). Hence, standard search strategies can traverse the VET because the structure of the ET is kept.

The VET is produced while running the program. For each invocation of a method, a new node is added to it with the method parameters and the context before the call. The result and the context after the call are only added to the node when the execution of the method finishes.

Let us explain the components of the architecture with the diagram in Figure 5.1. Observe that each tier contains a cache that can be seen as a view of the VET. Each cache is used for a different task:

Persistence cache. It is used to store the nodes of the VET in a database. Therefore, when the whole VET is in the database, the persistence cache is not used anymore. Basically, it specifies

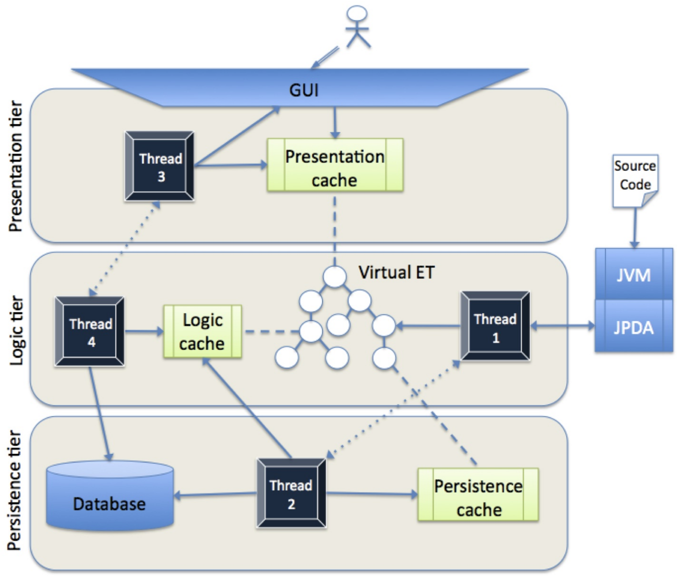


Figure 5.1: Architecture of a scalable algorithmic debugger

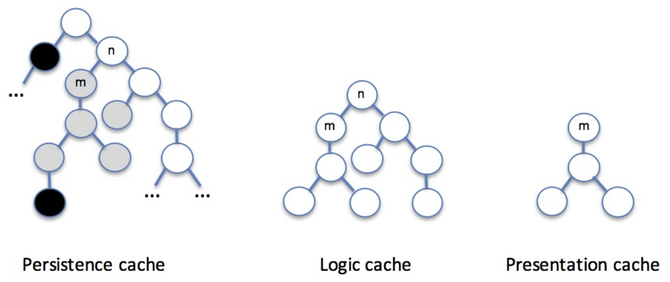
the maximum number of completed nodes that can be stored in the VET. This bound is called *persistence bound* and it ensures that the main memory is never overflowed.

Logic cache. It defines a subset of the VET. This subset contains a limited number of nodes (in the following, *logic bound*), and these nodes are those with the highest probability of being asked about, therefore, they should be retrieved from the database. This allows us to load in a single database transaction those nodes that are going to be probably asked and thus reducing the number of accesses to the database.

Presentation cache. It contains the part of the VET that is shown to the user in the GUI. The presentation cache defines a subtree inside the logic cache. Therefore, all the nodes in the presentation cache are also nodes of the logic cache. Here, the subtree is defined by selecting one root node and a depth (in the following, *presentation bound*).

The whole VET does not usually fit in main memory. Therefore, a mechanism to remove nodes from it and store them in a database is needed. When the number of complete nodes in the VET is close to the persistence bound, some of them are moved to the database, and only their identifiers remain in the VET. This allows the debugger to keep the whole ET structure in main memory and use identifiers to retrieve nodes from the database when needed.

Example 5.1.1 *Consider the following trees:*



The tree on the left is the VET of a debugging session where the grey nodes are those already completed (their associated method execution already finished); the black nodes are completed nodes that are only *stored in the database (only their identifiers are kept in the VET), and white nodes are nodes that have not been completed yet (they represent a method execution that has not finished yet). It may be possible that some of the white nodes have children not generated yet. Note that this VET is associated with an instant of the execution; and new nodes may be generated or completed later. The tree in the middle is the part of the VET referenced by the logic cache. In this case it is a tree, being n the root node and a depth of four, but in general it could contain unconnected nodes. Similarly, the tree on the right is the part of the VET referenced by the presentation cache (i.e., shown in the GUI), with m the root node and a depth of three. Note that the presentation cache is a subset of the logic cache.*

The behaviour of the debugger is controlled by four threads that run in parallel, one for the presentation tier (thread 3), two for the logic tier (threads 1 and 4) and one for the persistence tier (thread 2). Threads 1 and 2 control the generation of the VET and its storage in the database. They collaborate via synchronizations and message passing. Threads 3 and 4 communicate with the user and generate the questions. They also collaborate and are independent of threads 1 and 2. A description of the threads and their behaviour specified with pseudo-code follows:

Thread 1 (Construction of the VET) This thread is in charge of constructing the VET. It is the only one that communicates with the JPDA and JVM. Therefore, we could easily construct an algorithmic debugger for another language (e.g., C++) by only replacing this thread. Basically, this thread executes the program and for every performed method execution, it constructs and stores a new node in the VET. When the number of completed nodes (given by a function called *completedNodes*) is close to the persistence bound, this thread sends the *wake up* signal to thread 2. Then, thread 2 stores some nodes into the database.[2] If the persistence bound is reached, thread 1 sleeps until enough nodes have been removed from the VET and then it can continue generating new nodes.

Algorithm 2 Construction of the VET (Thread 1)

```
Input: A source program $\mathcal{P}$, and the persistence bound $persistenceBound$
Output: A VET $\mathcal{V}$
Initialization: $\mathcal{V} = (\emptyset, \emptyset)$

repeat
  1) run $\mathcal{P}$ inspecting the execution using JPDA and catch event $e$
  2) case $e$ of
  3)    new invocation of a method $I$:
  4)       create a new node $N$ from $I$
  5)       add $N$ to $\mathcal{V}$
  6)    the execution of the method related to $I$ ended:
  7)       complete node $N$ associated with $I$
  8)       if ($completedNodes(\mathcal{V}) == persistenceBound/2$) then
  9)          send thread 2 the wake up signal
 10)       if ($completedNodes(\mathcal{V}) == persistenceBound$) then
 11)          sleep
until the execution of $\mathcal{P}$ finishes or a bug is found
```

Thread 2 (Controlling the size of the VET) This thread ensures that the VET always fits in main memory. It controls which nodes of the VET should be stored in main memory, and which nodes should be stored in the database.

[2]In our implementation, it moves half of the nodes. Our experiments reveal that this is a good choice because it keeps threads 1 and 2 continuously running in a producer-consumer manner.

When the number of completed nodes in the VET is close to the persistence bound, thread 1 wakes thread 2 up, which removes some nodes from the VET and copies them into the database. It uses the logic cache to decide which nodes to store in the database. Concretely, it tries to store in the database as many nodes as possible that are not in the logic cache. When it finishes, it sends the *wake up* signal to thread 1 and sleeps.

Algorithm 3 Controlling the size of the VET (Thread 2)

```
Input: A VET $\mathcal{V}$
Output: An ET stored in a database

repeat
  1) sleep until wake up signal is received
  2) repeat
  3)    look into the persistence cache for the next completed node $N$ of the VET
  4)    if $N$ is not found then
  5)       wake thread 1 up
  6)    else
  7)       store $N$ in the database
  8)    if $N$ is not found or $N$ is the root node then
  9)       break
until the whole VET is stored in the database or until a bug is found
```

Thread 3 (Interface communication) This thread is the only one that communicates with the user. It controls the information shown in the GUI with the presentation cache. According to the user's answers, the selected strategy, and the presentation bound, this thread selects which will be the root node of the presentation cache. This task is done after each question according to the user's answers, ensuring that (i) the node that contains the next question (using a function called *AskQuestion*), (ii) its parent, and (iii) as many descendants as the presentation bound requires, are shown in the GUI.

Algorithm 4 Interface communication (Thread 3)

```
Input: User's answers
Output: A buggy node

repeat
  1) ask thread 4 to select a $node$
  2) update presentation cache and GUI visualization
  3) $answer = AskQuestion(node)$
  4) send $answer$ to thread 4
until a buggy node is found
```

Thread 4 (Selecting questions) This thread chooses the next node according to a given search strategy using a function called *SelectNextNode*. If the selected node is not loaded in the logic cache (*not(InLogicCache(node))*) the logic cache is updated. This is done by using the *UpdateLogicCache* function, which uses the selected node and the logic bound to compute the new logic cache (i.e, those nodes of the VET that should be loaded from the database). All the nodes that belong to the new logic cache and that do not belong to the previous logic cache are loaded from the database using the *FromDatabaseToET* function.

Algorithm 5 Selecting questions (Thread 4)

```
Input: A search strategy $\mathcal{S}$, a VET $\mathcal{V}$ and the logic bound $logicBound$
Output: A buggy node

repeat
  1) $node = SelectNextNode(\mathcal{V}, \mathcal{S})$
  2) if $not(InLogicCache(node))$ then
  3)     $missingNodes = UpdateLogicCache(node, logicBound)$
  4)     $\mathcal{V} = FromDatabaseToET(\mathcal{V}, missingNodes)$
  5) send $node$ to thread 3
  6) get $answer$ from thread 3 and change the state of the affected nodes
until a buggy node is found
```

An interesting task of this thread is carried out by the *UpdateLogicCache* function that determines what nodes should remain in the VET when a new node is selected by the search strategy. This task is not critical for the performance of the debugger. The bottleneck is the construction of the VET; contrarily, the exploration of the VET is a quick task that can be done step by step after each user's answer. However, the caching policy should be efficient to optimize the resources of the debugger, and to minimize the number of accesses to the database. In particular, those nodes that are already cached in the VET and that could be needed in future questions should remain in the VET. Contrarily, those nodes that are definitely discarded by the user's answers should be replaced by other nodes that can be useful.

Although each search strategy uses a different caching policy, all of them share four invariants:

1. When a node is marked as wrong, then all the nodes that are not descendants of this node are useless, and they can be unloaded from the VET.

2. When a node is marked as right, then all the descendants of this node are useless, and they can be unloaded from the VET.

3. When a node is marked as wrong, then all the descendants of this node that have not been discarded yet, could be needed in the future, and they remain in the VET.

4. When a node is marked as right, then all the nodes that are not descendants of this node (and that have not been discarded yet) could be needed in the future, and they remain in the VET.

As an example, with the Top-Down strategy, initially, we load in the VET the root node and those nodes that are close to the root. Every time a node is marked as correct, all the nodes in the subtree rooted at this node are marked as useless and they can be unloaded from the VET when it is needed. All the nodes that have been already answered, except the last node marked as wrong can be also unloaded. When the search strategy is going to select a node that is not loaded in the VET, then all the useless nodes are unloaded from the VET, and those nodes that are descendant of the selected node and that are closer to it are loaded. In this way, the number of accesses to the database are minimized, because every load of nodes to the VET provides enough nodes to answer several questions before a new load is needed.

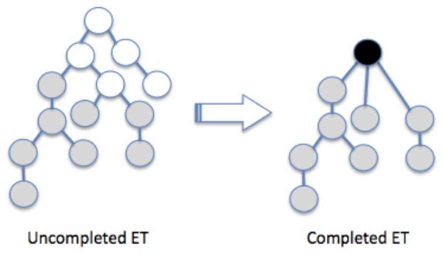
Redefining the search strategies for Algorithmic Debugging

In Algorithm 5, a search strategy is used to generate the sequence of questions by selecting nodes from the VET. Nevertheless, all AD search strategies in the literature have been defined for ETs and not for VETs where incomplete nodes may exist. All of them assume that the information of all ET nodes is available. Clearly, this is not true in our context and thus, the search strategies would fail. For instance, the first node asked by the Top-Down strategy and its variants is always the root node of the ET. However, this node is the last node completed by Algorithm 2. Hence, these search strategies could not even start until the whole ET is completed, and this is exactly the problem that we want to solve.

Therefore, in this section we propose a redefinition of the search strategies of AD so that they can work with VETs.

A first solution could be to define a transformation from a VET with incomplete nodes to a VET where all nodes are completed. This can be done by inserting a new root node with the equation $1 = 0$. Then, the children of this node would be those completed nodes whose parent is incomplete. In this way, (i) all nodes of the produced ET would be completed and could be asked; (ii) the parent-child relation is kept in all the subtrees of the ET; and (iii) it is guaranteed that at least one bug (the root node) exists. If the debugging session finishes with the root node as buggy, it means that the node with the "real" bug (if any) has not been completed yet.

Example 5.1.2 *Consider the following VETs:*



In the VET on the left, the grey nodes have been completed, and the white nodes are incomplete. This VET can be transformed into the VET on the right where all nodes are completed. The new artificial root is the black node which ensures that at least one buggy node exists.

From an implementation point of view, this transformation is inefficient and costly because the VET is continuously being generated by thread 1, and hence, this transformation should be done repeatedly question after question. In contrast, a more efficient solution is to redefine the search strategies so that they ignore incomplete nodes. For instance, Top-Down [2] would only ask first about those nodes that are completed and that do not have a completed ancestor. Similarly, Binks' top-down [5] would ask first about the heaviest nodes of these nodes. D&Q [92] would ask about the completed node that divides the VET into two subtrees with the same number of nodes, and so on. We refer the interested reader to the source code of our implementation that is publicly available and where all search strategies have been reimplemented to deal with VETs.

Even though the architecture presented has been discussed in the context of Java, it can work in other languages with very few changes. Observe that the part of an algorithmic debugger that is language-dependent is the front-end, and our technique relies on the back-end. Once the VET is generated, the back-end can handle the VET mostly independent of the language. We provide a discussion of the changes needed to adapt the architecture to other languages in the next section.

5.1.3 Implementation

We have implemented the proposed technique and integrated it into DDJ 2.4. The implementation has been tested with a collection of real applications. Table 5.1 summarizes the results of the experiments. These experiments have been done in an Intel Core2 Quad 2.67 GHz with 2GB RAM. The first column contains the names of the benchmarks. For each benchmark, the second and third columns give an idea of the size of the execution. Fourth and fifth columns are time measures. Finally, sixth and seventh columns show memory bounds. Concretely, the `variables number` column shows the number of variables participating (possibly repeated) in the considered execution. It is the sum of all variables in all the nodes of the ET. The `ET size` column shows the total size in MB of the ET when it is completed, this measure has been taken from the size of the ET in the database. The `ET time` column is the time needed to completely generate the whole ET. The `node time` column is the time needed to complete the first node of the ET. The `cache limit` column shows the depth of the logic cache of these benchmarks. After these bounds, the computer was out of memory. Finally, the `ET depth` column shows the depth of the ET after it was constructed.

Observe that a standard algorithmic debugger is hardly scalable to these real programs. With the standard technique, even if the ET fits in main memory or we use a database, the user must wait for a long time until the ET is finally completed and the first question can be asked. In the worst case, this time is more than one hour. Contrarily, with the new technique, the debugger can start to ask questions before the ET is completed. Note that the time needed to complete the first node is always less than two seconds. Therefore, the debugging session can start almost instantaneously.

All the information related to the experiments, the source code of the benchmarks, the bugs, the source code of the tool and other material can be found at:

`http://www.dsic.upv.es/~jsilva/DDJ/`

Adapting the architecture for other languages

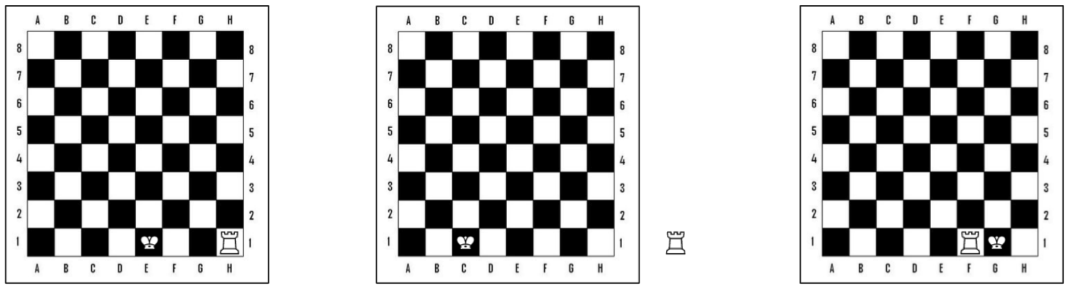
Even though the presented technique is based on the Java language, it is mostly independent of the language that is being debugged. Note that all algorithmic debuggers are based on the ET and the main difference between debuggers from different languages is the information stored in each node (e.g., functions in the functional paradigm, methods in the object-oriented paradigm, functions and procedures in the imperative paradigm, predicates in the logic paradigm, etc.). But once the ET is constructed, the search strategies that traverse the ET are independent of the language that the ET represents.

Therefore, because our technique is based on the structure of the ET, but not on the information of its nodes, all the components of the architecture presented in Figure 5.1 could be used for other languages with small changes. In the case of other object-oriented languages such as C++ or C#, the only component that needs to be changed is thread 1. In other languages, the required changes would involve the redefinition of the components to change the kind of information (e.g., clauses, predicates, procedures, etc.) stored (in the database and in the virtual ET) and shown (in the GUI). However, the behaviour of all the components and of all the threads would be exactly the same. The only part of the architecture that should be completely changed is the part in charge of the construction of the ET. JPDA is exclusive for Java, hence, thread 1 should be updated to communicate with another component similar to the JPDA but for a different language. All the interrelations among the components would remain unchanged. Finally, another small modification would be done in the GUI. Our implementation shows objects, methods and attributes. Showing procedures, global variables or functions would be very similar, but small changes would still be needed.

5.2 Balancing Execution Trees

5.2.1 Introduction

Balancing Execution Trees [55] (also referred to as Tree Balancing) is a technique of Algorithmic Debugging (AD) that changes the structure of the Execution Tree (ET) in such a way that the search strategies present an almost optimal behaviour. The objective of the technique is to balance[3] the ET in such a way that search strategies prune half of the ET at every step, because they can always find a node that divides the search area in half. Our experiments with real programs show that the technique reduces (as an average) the number of questions to the user by around 30 %. Without loss of generality, along this section we base our examples on programs implemented using the Java language, although our ET transformations and the balancing technique are conceptually applicable to any ET, regardless of the language the source code that generates the ET is written in.



Initial position Final position Expected position

```
public class Chess {
    public static void main(String[] args) {
        Chess chess = new Chess();
        Position king = new Position();
        Position rook = new Position();

        king.locate(5, 1);
        rook.locate(8, 1);
        chess.castling(king, rook);
    }

    void castling(Position k, Position r) {
        if (r.x != 8) { // Bug!!
        // Correct: if (r.x == 8) {
            for (int i = 1; i <= 2; i++) { k.right(); }
            for (int i = 1; i <= 2; i++) { r.left(); }
        } else {
            for (int i = 1; i <= 2; i++) { k.left(); }
            for (int i = 1; i <= 3; i++) { r.right(); }
        }
    }
}

public class Position {
    int x, y;
    void locate(int a, int b) { x = a; y = b; }
    void up() { y = y + 1; }
    void down() { y = y - 1; }
    void right() { x = x + 1; }
    void left() { x = x - 1; }
}
```

Figure 5.2: Chess Java program

[3]Note that throughout this section we use *balanced* to indicate that the tree becomes *binomial*.

Example 5.2.1 *Consider the Java program in Figure 5.2. This program has a bug, and thus it wrongly simulates a movement on a chessboard. The* `chess.castling(king, rook)` *call produces the (wrong) movement shown in the chessboards of the figure. Figure 5.3 depicts the portion of the ET associated with the* `chess.castling(king, rook)` *call. With this ET, all current search strategies need to ask about the six nodes. In contrast, if we balance this ET with our transformation, the bug is found with three questions at the most.*

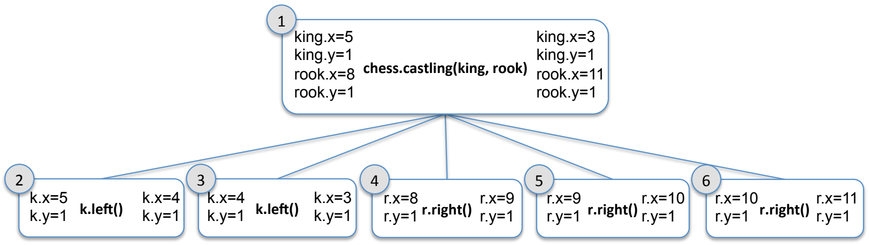

1
king.x=5
king.y=1
rook.x=8
rook.y=1
chess.castling(king, rook)
king.x=3
king.y=1
rook.x=11
rook.y=1
2
k.x=5 k.y=1 k.left() k.x=4 k.y=1
3
k.x=4 k.y=1 k.left() k.x=3 k.y=1
4
r.x=8 r.y=1 r.right() r.x=9 r.y=1
5
r.x=9 r.y=1 r.right() r.x=10 r.y=1
6
r.x=10 r.y=1 r.right() r.x=11 r.y=1

Figure 5.3: ET associated with the `chess.castling(king, rook)` call of the program in Figure 5.2

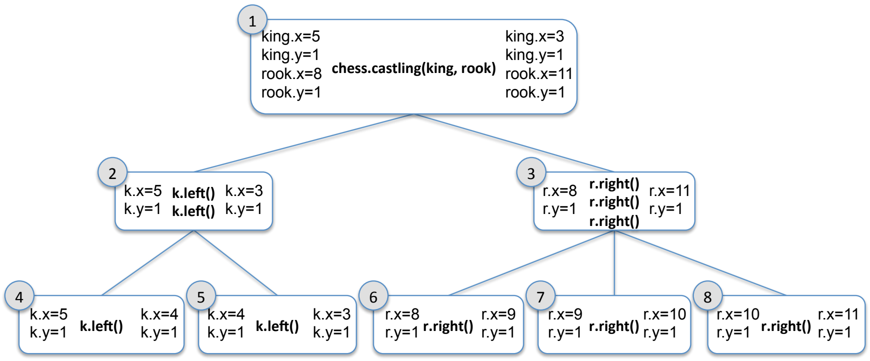

1
king.x=5
king.y=1
rook.x=8
rook.y=1
chess.castling(king, rook)
king.x=3
king.y=1
rook.x=11
rook.y=1
2
k.x=5 k.y=1 k.left() k.left() k.x=3 k.y=1
3
r.x=8 r.y=1 r.right() r.right() r.right() r.x=11 r.y=1
4
k.x=5 k.y=1 k.left() k.x=4 k.y=1
5
k.x=4 k.y=1 k.left() k.x=3 k.y=1
6
r.x=8 r.y=1 r.right() r.x=9 r.y=1
7
r.x=9 r.y=1 r.right() r.x=10 r.y=1
8
r.x=10 r.y=1 r.right() r.x=11 r.y=1

Figure 5.4: Balanced ET associated with the `chess.castling(king, rook)` call in Figure 5.2

Tree Balancing presents three important advantages that make it useful for AD. First, it can be easily adapted to other programming languages. We have implemented it for Java and it can be directly used in other object-oriented languages, but it could be easily adapted to other languages such as C or Haskell using the analogy between methods and functions. Second, the technique is quite simple to implement and can be integrated into any existing algorithmic debugger with small changes. And third, the technique is conservative. If the questions created by the new nodes are difficult to answer, the user can answer "I don't know" and continue the debugging session as in the standard ET. Moreover, the user can naturally get back to the original ET, if need be.

The rest of this section has been organized as follows. In Section 5.2.2 we introduce some preliminary definitions that will be used in the rest of the section. In Section 5.2.3 we explain in detail tree balancing and its main applications, and we introduce the algorithms used to balance ETs. The correctness of the technique is described in Section 5.2.4. Then, in Section 5.2.5, we present our implementation and some experiments carried out with real Java programs. Section 5.2.6, discusses the related work. The proofs of correctness are shown in Section 5.2.7. Finally, Section 5.2.8 describes a case of study.

5.2.2 Preliminary definitions

In this section we introduce the new ET nodes created by Tree Balancing as well as the intended semantics that is used to answer the questions of these nodes. We start with the definition of *Composite Method Execution*:

Definition 5.2.2 (Composite Method Execution) *A* composite method execution *is a non-empty sequence of method executions* $\langle(b_1, m_1, e_1), (b_2, m_2, e_2), \ldots, (b_n, m_n, e_n)\rangle$ *that we represent using the notation* $(b_1; m_1; m_2; \ldots; m_n, e_n)$.

From now on, we will assume that there exists an intended semantics $\mathcal{I}$ of the program being debugged. It corresponds to the model the programmer had in mind while writing the program, and it contains, for each method m and each context b of m at the beginning of its execution, the expected context e at the end of its execution, that is, $(b, m, e) \in \mathcal{I}$. Moreover, given this atomic information, we are able to deduce judgments of the form $(b, m_1; \ldots; m_n, e)$ with the inference rule Tr, that defines the transitivity of the composition of methods

$$\frac{\overline{(b, m_1, e')} \quad \overline{(e', m_2; \ldots; m_n, e)}}{(b, m_1; \ldots; m_n, e)}\ \mathsf{Tr}\ \text{if}\ n > 1$$

and we say that $\mathcal{I} \models (b, m_1; \ldots; m_n, e)$. Using this intended semantics we can formally define the correctness of method executions.

Definition 5.2.3 (Method Execution correctness) *Given a method execution* $\mathcal{E}$ *and the intended semantics of the program* $\mathcal{I}$*, we say that* $\mathcal{E}$ *is* correct *if* $\mathcal{E} \in \mathcal{I}$ *or* $\mathcal{I} \models \mathcal{E}$ *and* wrong *otherwise.*

Example 5.2.4 *An AD session for the ET in Figure 5.3 using D&Q follows (*`YES` *and* `NO` *answers are provided by the user):*

```
Starting Debugging Session...
(2)  k.x = 5,  k.y = 1  >>>  k.left()   >>>  k.x = 4,  k.y = 1 ?      YES
(3)  k.x = 4,  k.y = 1  >>>  k.left()   >>>  k.x = 3,  k.y = 1 ?      YES
(4)  r.x = 8,  r.y = 1  >>>  r.right()  >>>  r.x = 9,  r.y = 1 ?      YES
(5)  r.x = 9,  r.y = 1  >>>  r.right()  >>>  r.x = 10, r.y = 1 ?      YES
(6)  r.x = 10, r.y = 1  >>>  r.right()  >>>  r.x = 11, r.y = 1 ?      YES
(1)  king.x = 5,                                     king.x = 3,
     king.y = 1,  >>>  chess.castling(king, rook)  >>>  king.y = 1, ?  NO
     rook.x = 8,                                     rook.x = 11,
     rook.y = 1                                      rook.y = 1
Bug located in method: castling(Position t, Position k) of the Chess class.
```

The debugger points out the buggy method, which contains the bug. In this case, `r.x != 8` *should be* `r.x == 8`.

5.2.3 Collapsing and projecting nodes

Even though the Heaviest First strategy significantly improves the Top-Down search, its performance strongly depends on the structure of the ET. The more balanced the ET is, the better. Clearly, when the ET is balanced, Heaviest First is much more efficient because it prunes more nodes after every question. If the ET is completely balanced, Heaviest First is equivalent to Divide and Query and both are query-optimal.

Advantages of collapsing and projecting nodes

Tree Balancing is based on a series of transformations that allows for collapsing/projecting some nodes of the ET. A collapse node is a new node that replaces some nodes that are then removed from the ET. In contrast, a projection node is a new node that is placed as the parent of a set of nodes that remain in the ET. This section describes the main advantages of collapsing/projecting nodes:

Balancing execution trees. If we augment an ET with projection nodes, we can strategically place them in such a way that the ET becomes balanced. In this way, the debugger speeds up the debugging session by reducing the number of asked questions.

Example 5.2.5 *Consider again the program in Figure 5.2. The portion of the ET associated with* `chess.castling(king, rook)` *is shown in Figure 5.3. We can add projection nodes to this ET as depicted in Figure 5.4. Note that now the ET becomes balanced, and hence, many search strategies ask fewer questions. For instance, in the worst case, using the ET of Figure 5.3 the debugger would ask about all the nodes before the bug is found. This is due to the broad nature of this ET that prevents search strategies from pruning any nodes. In contrast, using the ET of Figure 5.4 the debugger prunes almost half of the tree with every answer. In this example, with the standard ET of Figure 5.3, D&Q produces the following debugging session (numbers refer to the identifiers of the nodes in the figure):*

```
Starting Debugging Session...
(2) YES  (3) YES  (4) YES  (5) YES  (6) YES  (1) NO
Bug located in method: castling(Position k, Position r) of the Chess class.
```

In contrast, with the ET in Figure 5.4, D&Q produces this session:

```
Starting Debugging Session...
(2) YES  (3) YES  (1) NO
Bug located in method: castling(Position k, Position r) of the Chess class.
```

Skipping repetitive questions. Algorithmic debuggers tend to repeat the same (or very similar) question several times when it is associated with a method execution performed inside a loop. In our example, this happens in `for (int i = 1; i <= 3; i++) { r.right(); }`, which is used to move the rook three positions to the right. Here, the nodes

```
{ r.x = 8,  r.y = 1 }  r.right()  { r.x = 9,  r.y = 1 }
{ r.x = 9,  r.y = 1 }  r.right()  { r.x = 10, r.y = 1 }
{ r.x = 10, r.y = 1 }  r.right()  { r.x = 11, r.y = 1 }
```

could be projected to the node

```
{ r.x = 8, r.y = 1 }  r.right(); r.right(); r.right()  { r.x = 11, r.y = 1 }
```

This kind of projection, where all the projected nodes refer to the same method, has an interesting property: If the projected nodes are leaves, then they can be deleted from the ET. The reason is that the new projection node (node 2 in Figure 5.4) and the projected nodes (nodes 4 and 5 in Figure 5.4) refer to the same method. Therefore, it does not matter which one is the buggy node, because the bug will necessarily be in this method. Hence, if the projection node is wrong, then the bug is in the method pointed to by this node. When the children of the projection node are removed, we call it *collapse node*.

Note that, in this case, the idea is not to add nodes to the ET as in the previous case, but to delete them. Because the input and output of all the questions relate to the same attributes (i.e., x and y), then the user can answer them all together, since they are, in fact, a sequence of operations whose output is the input of the next question (i.e., they are chained). Therefore, this technique allows for treating a set of questions as a whole. This is particularly interesting because it approximates the real behaviour intended by the programmer. For instance, in this example, the intended meaning of the loop was to move the rook three positions to the right. The intermediate positions are not interesting; only the initial and final ones are meaningful for the intended meaning.

Example 5.2.6 *Consider the ET of Figure 5.4. Observe that, if the projected nodes are wrong, then the bug must be in the unique method appearing in the projection node. Thus, we could collapse the node instead of projecting it. Hence, nodes 4, 5, 6, 7, and 8 could be removed; and thus, with only three questions we could discover any bug wherever it is in the code.*

Enhancing the search of algorithmic debugging. One important problem of AD search strategies is that they must use a given ET without any possibility of changing it. This often prevents search strategies from selecting nodes that prune a big part of the ET, or from selecting nodes that are in the regions with a higher probability of containing the bug. Encapsulating some subtrees into a single node can help to solve these drawbacks.

The initial idea of this section was to use projection nodes to balance the ET. This idea is very interesting in combination with D&Q, because it can cause the debugging session to be optimal in the worst case (its query complexity is $O(b \cdot log\ n)$, where b is the branching factor and n is the number of nodes in the ET). However, this idea could be further extended in order to force the search strategies to ask questions related to parts of the computations with a higher probability of containing the bug. Concretely, we can replace parts of the ET with a capsule node in order to avoid questions related to this part. If the debugging session determines that the capsule node is wrong, we can expand it again to continue the debugging session inside this node. Therefore, with this idea, the original ET is transformed into a tree of ETs that can be explored when it is required. Let us illustrate this idea with an example.

Example 5.2.7 *Consider the leftmost ET in Figure 5.5. This ET has a root that started two subcomputations. The computation on the left performed ten method executions, while the computation on the right performed only three. Hence, in this ET, all the existing search strategies would explore first the left subtree.*[4] *If we balance the left branch by inserting projection nodes we obtain the new ET shown in the middle. This balanced ET requires (on average) fewer questions than the previous one; but the search strategies will still explore first the left branch of the root.*

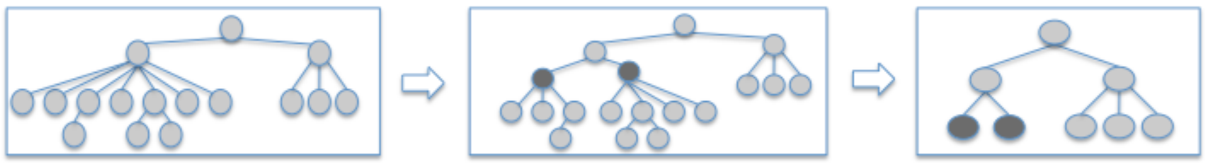


Figure 5.5: Balancing transformations of ETs

Now, let us assume that the debugger identified the right branch as more likely to be buggy (e.g., because it contains recursive calls, because it is non-deterministic, because it contains calls with more involved arguments or with complex data structures...). We can change the structure of the ET in order to make AD search strategies to start exploring the desired branch. In this example, we can encapsulate the projected subtrees. The new ET is shown on the right of Figure 5.5. With this ET, the tool explores first the right branch of the root. Note that these nodes are similar to the collapse nodes in terms of removing their children from the tree, but observe that in this case it is not necessary that the nodes that were projected refer to the same method. They can be completely different and independent computations. However, if the debugger determines that they are probably correct, they can be encapsulated to direct the search to other parts of the ET. Of course, they can be decapsulated again if required by the search strategy (e.g., if the debugger cannot find the bug in the other nodes).

Disadvantages Given these benefits, we must talk about a potential drawback: the difficulty of the questions related to the new nodes. The new questions may be more difficult to answer than the previous ones, but the user can avoid these difficult questions by answering “I don’t know”. In such a case the algorithmic debugger would behave as if the projection/collapse node did not exist, and hence as if the technique would have never been applied. However, the nodes that were used to create the new projection/collapse node are intimately related, as it can be seen in Definitions 5.2.9, 5.2.10, and 5.2.11. For instance, during the execution of loops many projection nodes are generated by projecting the nodes that have been produced along different iterations. Thus, in many situations the question generated

[4]Current search strategies assume that all nodes have the same probability of being buggy, therefore, heavy branches are explored first.

from the projection/collapse node is more related to the behaviour the programmer had in mind when writing the code than the original questions (i.e., the behaviour of the whole loop vs. the behaviour of each single call inside the loop).

Algorithms for collapsing and projecting nodes

In this section we define a technique that allows for balancing an ET while keeping the soundness and completeness of AD. The technique is based on two basic transformations for ETs (namely *collapse leaf chain* and *project chain*, described respectively in Algorithms 6 and 7), and on a new data structure called *Execution Forest* (EF) that is a generalization of an ET.

Definition 5.2.8 (Execution Forest) *An* execution forest *is a tree $T = (N, E)$ whose internal nodes N are method executions and whose leaves are either method executions or execution forests.*

Roughly speaking, an EF is an ET where some subtrees have been replaced (i.e., encapsulated) by a single node. Note that this recursive definition of EF is more general than the one for ET because an ET is an instance of an EF where no encapsulated nodes exist. We can now define the two basic transformations of our technique. Both transformations are based on the notion of *chain*. Informally, a chain is formed by an ordered set of sibling nodes in which the final context produced by a node of the chain is the initial context of the next node. Chains often represent a sequence of method executions performed one after the other during an execution. Formally,

Definition 5.2.9 (Chain) *Given an EF $T = (N, E)$ and a set $C \subset N$ of $k \geq 2$ nodes with associated method executions $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_k$ we say that C is a* chain *iff*

- *$\exists n \in N$ such that $\forall c \in C.(n \to c) \in E$,*
- *$\forall i, j, 1 \leq i < j \leq k$, the method associated with $\mathcal{E}_i$ is executed before the method associated with $\mathcal{E}_j$, and*
- *$\forall j, 1 \leq j \leq k-1$, if $\mathcal{E}_j = (e_j, m_j, e_{j+1})$ then $\mathcal{E}_{j+1} = (e_{j+1}, m_{j+1}, e_{j+2})$*

The first condition ensures that all the elements in the chain are siblings. The second condition ensures that the elements are executed one after the other. The third condition ensures that for all the nodes in the chain the final context of a node is the initial context of the next chained node. Note that, by Definition 4.1.1, only those attributes that can be affected by the execution of the methods are taken into account. It is common to find chains when one or more methods are executed inside a loop.

Now we have defined chains we can provide a definition of projection and collapse nodes.

Definition 5.2.10 (Projection node) *Given an EF $T = (N, E)$, and a composite method execution $n = (b, m_1; m_2; \ldots; m_k, e) \in N$. n is a projection node iff there exists a chain C such that:*

- *$C = \{(b, m_1, e_1), (e_1, m_2, e_2), \ldots, (e_{k-1}, m_k, e)\}$*
- *$\forall c \in C, (n \to c) \in E$, and*
- *$\forall n' \in N$ such that $(n \to n') \in E \ . \ n' \in C$*

Definition 5.2.11 (Collapse node) *Given an EF $T = (N, E)$, and a composite method execution $n = (b, m_1; m_2; \ldots; m_k, e) \in N$. n is a collapse node iff there exists a chain C such that:*

- *$C = \{(b, m_1, e_1), (e_1, m_2, e_2), \ldots, (e_{k-1}, m_k, e)\}$, and*

Algorithm 6 Collapse Leaf Chain.

```
**Input:** An EF $T = (N, E)$ and a set of nodes $C \subset T$
**Output:** An EF $T' = (N', E')$
**Preconditions:** $C$ is a chain with nodes $(a_1, m, a_2), (a_2, m, a_3), \ldots, (a_k, m, a_{k+1})$ and $\nexists n \in N.(c \rightarrow n) \in E$, with $c \in C$

begin
  1) $parent = p \in N$ such that $\forall c \in C, (p \rightarrow c) \in E$
  2) $colnode = (a_1, m', a_{k+1})$ with $m' = m; m; \ldots; m$
  3) $N' = (N \backslash C) \cup \{colnode\}$
  4) $E' = (E \backslash \{(parent \rightarrow n) \in E \mid n \in C\}) \cup \{(parent \rightarrow colnode)\}$
  5) return $T' = (N', E')$
end
```

Algorithm 7 Project Chain.

```
**Input:** An EF $T = (N, E)$ and a set of nodes $C \subset N$
**Output:** An EF $T' = (N', E')$
**Preconditions:** $C$ is a chain with nodes $(a_1, m_1, a_2), (a_2, m_2, a_3), \ldots, (a_k, m_k, a_{k+1})$

begin
  1) $parent = p \in N$ such that $\forall c \in C, (p \rightarrow c) \in E$
  2) $prjnode = (a_1, m, a_{k+1})$ with $m = m_1; m_2; \ldots; m_k$
  3) $N' = N \cup \{prjnode\}$
  4) $E' = (E \backslash \{(parent \rightarrow n) \in E \mid n \in C\}) \cup \{(parent \rightarrow prjnode)\} \cup \{(prjnode \rightarrow c) \mid c \in C\}$
  5) return $T' = (N', E')$
end
```

- $\nexists n' \in N$ *such that* $(n \rightarrow n') \in E$

The basic transformations of chains are described in Algorithms 6 and 7. Algorithm 6 is in charge of collapsing chains, which consists in creating a new node *colnode* with initial context the initial context of the first node of the chain, final context the final context of the last node of the chain, and with the composition of the methods in the chain as associated method. Then, the nodes in the chain (and thus their edges) are removed from the tree and the new node is added as a child of the parent of the nodes in C, thus reducing the size of the EF. Algorithm 7 is in charge of projecting chains, and works in a similar way as Algorithm 6. Given a tree T and a chain C in the tree, it removes from T the edges between each $c \in C$ and its parent *parent*, and then introduces a new node *prjnode* built as explained before, that is linked to each c as their new parent, and to *parent* as its new child.

Algorithm 8 is in charge of removing the chains of leaves that can be collapsed. It first computes in the initialization all the maximal chains (i.e., chains that are not subchains of other chains) of nodes that are leaves and are related to the same methods. Then, for each of these chains, it applies Algorithm 6 to collapse them by removing the chain from the tree and adding the corresponding collapse node.[5]

Our method for balancing EFs is implemented in Algorithm 9. This algorithm first uses Algorithm 8 to shrink the EF (line 1) by collapsing as many nodes as possible; and then it balances this shrunken EF by projecting some nodes. The objective is to divide the tree into two parts with the same weight (i.e., number of nodes). Therefore, we first compute half of the size of the EF (lines 4 and 5). If a

[5]Note that, since these chains are usually found when a loop or a recursive call is used, our approach generates nodes whose questions are very close to the intended meaning the programmer had in mind while developing the program and thus, although the new questions comprise a bigger context, they may be even easier to answer than the "atomic" ones.

Algorithm 8 Shrink EF.

```
Input: An EF $T = (N, E)$
Output: An EF $T' = (N', E')$
Preconditions: Given a node $n$, $n.method$ is the name of the method in $n$
Initialization: $T' = T$, the $\mathcal{S}$ set contains all the maximal chains of $T$ s.t. for each chain $s = \{c_1, \ldots, c_k\}$ of $\mathcal{S}, \forall i, 1 \leq i \leq k-1, c_i.method == c_{i+1}.method$, and $\nexists n \in N.(c \to n) \in E$, with $c \in s$

begin
  1) while ($\mathcal{S} \neq \emptyset$)
  2)     take a chain $s \in \mathcal{S}$
  3)     $\mathcal{S} = \mathcal{S} \backslash \{s\}$
  4)     $T' = collapseChain(T', s)$
  5) end while
  6) return $T'$
end
```

child of the root is already heavier than half the size of the tree, then, the weight of this node is not taken into account in the balancing process because the question associated to this node will be the first question to be asked (lines 9-16). Otherwise, it projects the part of a chain whose weight is as close as possible to half the weight of the root (lines 17-26). This allows for pruning half of the subtree when asking a question associated with a projection node. In the case that the heaviest node (lines 13-16) or the projected chain (lines 23-24) belongs to a bigger chain, it must be cut with the *cutChain* function producing new (smaller) chains that are also processed. Of course, the size of the already processed chains is not taken into account when dividing the successive (sub)chains (because they will be already pruned during a debugging session).

If a chain is very long, it can be cut in several subchains to be projected and thus better balance the EF. In order to cut chains we use the *cutChain* function:

```
function $cutChain$(chain $\{c_1, \ldots, c_k\}$, int $i$, int $j$)
  if ($i > 2$) then $s_{ini} = \{c_1, \ldots, c_{i-1}\}$ else $s_{ini} = \emptyset$ end if
  if ($k - j > 1$) then $s_{end} = \{c_{j+1}, \ldots, c_k\}$ else $s_{end} = \emptyset$ end if
return ($s_{ini}, s_{end}$)
```

This function removes from a chain a subchain delimited by indices i and j. As a result, depending on the indices, it can produce two subchains that are located before and after the subchain. Note that when the initial index i is 2, there is only one node remaining before the subchain, and thus, because it is not a chain, $\emptyset$ is returned. The same happens on the right.

The algorithm finishes when no more chains can be projected. Trivially, because the number of chains and their length are finite, termination is ensured. In addition, Algorithm 9 is able to balance the EF while it is being computed. Concretely, the algorithm should be executed for each node of the EF that is completed (i.e., the final context of the method execution is already calculated, thus all the children of this node are also completed). Note that this means that the algorithm is applied bottom-up to the nodes of the EF. Hence, when balancing a node, all the descendants of the node have already been balanced. This also means that modern debuggers that are able to debug programs with uncompleted ETs (see Section 5.1) can also use the technique, because the ET can be balanced while it is being computed.

Algorithm 9 Shrink & Balance EF.

```
Input: An EF $T = (N, E)$ whose root is $root \in N$
Output: An EF $T' = (N', E')$
Preconditions: Given a node $n$, $n.weight$ is the size of the subtree rooted at $n$

begin
  1) $T' = shrink(T)$
  2) $children = \{n \in N' \mid (root \to n) \in E'\}$
  3) $\mathcal{S} = \{s \mid s$ is a maximal chain in $children\}$
  4) $rootweight = root.weight$
  5) $half = rootweight/2$
  6) while $(\mathcal{S} \neq \emptyset)$
  7)    $child = c \in children$ s.t. $\nexists c' \in children, c'.weight > c.weight$
  8)    $distance = |half - child.weight|$
  9)    if $(child.weight \geq half \vee \nexists i, j$ s.t. $\{c_1, \ldots, c_k\} \in S \wedge (|half - \sum_{x=i}^{j} c_x.weight| < distance))$ then
 10)       $children = children \backslash \{child\}$
 11)       $rootweight = rootweight - child.weight$
 12)       $half = rootweight/2$
 13)       if $(\exists s \in \mathcal{S}$ s.t. $s = \{c_1, \ldots, c_k\}$ and $child = c_i, 1 \leq i \leq k)$ then
 14)          $(s_{ini}, s_{end}) = cutChain(s, i, i)$
 15)          $\mathcal{S} = (\mathcal{S} \backslash \{s\}) \cup s_{ini} \cup s_{end}$
 16)       end if
 17)    else
 18)       find an $s, i, j$ s.t. $s = \{c_1, \ldots, c_k\} \in S$ and $\sum_{x=i}^{j} c_x.weight$ is as close as possible to $half$
 19)       $s' = \{c_i, \ldots, c_j\}$
 20)       $children = children \backslash s'$
 21)       $rootweight = rootweight - \sum\{c.weight \mid c \in s'\}$
 22)       $half = rootweight/2$
 23)       $(s_{ini}, s_{end}) = cutChain(s, i, j)$
 24)       $\mathcal{S} = (\mathcal{S} \backslash \{s\}) \cup s_{ini} \cup s_{end}$
 25)       $T' = projectChain(T', s')$
 26)    end if
 27) end while
 28) return $T' = (N', E')$
end
```

5.2.4 Correctness

Our technique for balancing EFs is based on the transformations presented in the previous section. We present in this section the theoretical results about soundness and completeness, whose proofs are available in Section 5.2.7.

Theorem 5.2.12 (Completeness and soundness of EFs) *Given an EF with a wrong root, it contains a buggy node, which is associated with a buggy method.*

Completeness and soundness are kept after our transformations. In particular, an EF with a wrong root and a buggy node still has a buggy node after any number of collapses or projections.

Theorem 5.2.13 (Chain Collapse Correctness) *Let $T = (N, E)$ and $T' = (N', E')$ be two EFs, being the root of T wrong, and let $C \subset N$ be a chain such that all nodes in the chain are leaves and they have the same associated method. Given $T' =$* `collapseChain(T, C)`*,*

1. *T' contains a buggy node.*

2. *Every buggy node in T' is associated with a buggy method.*

Theorem 5.2.14 (Chain Projection Correctness) *Let $T = (N, E)$ and $T' = (N', E')$ be two EFs, and let $C \subset N$ be a chain such that $T' =$ `projectChain(T, C)`.*

1. *All buggy nodes in T are also buggy nodes in T'.*

2. *Every buggy node in T' is associated with a buggy method.*

We also provide in this section an interesting result related to the projection of chains. This result is related to the incompleteness of the technique when it is used intra-session (i.e., in a single debugging session trying to find one particular bug). Concretely, the following result does not hold: *A buggy node can be found in an EF if and only if it can be found in its balanced version.*

In general, our technique ensures that all the bugs that caused the wrong behaviour of the root node (i.e., the wrong final context of the whole program) can be found in the balanced EF. This means that all those buggy nodes that are responsible of the wrong behaviour are present in the balanced EF.

However, AD can find bugs just by chance. Those nodes that are buggy nodes in the EF but did not cause the wrong behaviour of the root node can be undetectable with some search strategies in the balanced version of the EF. The opposite is also true: *it is possible to find bugs in the balanced EF that were undetectable in the original EF.* Let us explain it with an example.

Example 5.2.15 *Consider the EFs in Figure 5.6. The EF on the right is the same as the one on the left but a new projection node has been added. If we assume the following intended semantics (expressed with triples of the form: initial context, method, final context) then the grey nodes are wrong and the white nodes are right:*

$$\begin{array}{llllllllllll} x=1 & g() & x=4 & \quad & x=4 & g() & x=4 & \quad & x=1 & f() & x=2 \\ x=3 & h() & x=3 & \quad & x=4 & h() & x=4 & & & & \end{array}$$

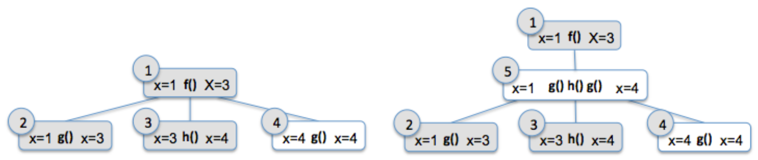


Figure 5.6: New buggy nodes revealed by the Tree Balancing technique

Note that in the EF on the left, only nodes 2 and 3 are buggy. Therefore, all the search strategies will report these nodes as buggy, but never node 1. However, node 1 contains a bug but it is undetectable by the debugger until nodes 2 and 3 have been corrected. Nevertheless, observe that nodes 2 and 3 did not produce the wrong behaviour of node 1. They simply produced two errors that, in combination, produced by chance a global correct behaviour.

Now, observe in the EF on the right that node 1 is buggy and thus detectable by the search strategies. In contrast, nodes 2 and 3 are now undetectable by the Top-Down search (they could be detected by D&Q). Thanks to the balancing process, it has been made explicit that three different bugs exist in the EF.

5.2.5 Implementation

We have implemented the presented technique and integrated it into an algorithmic debugger for Java. The implementation allows the user to activate the transformations of the technique and to parameterize them in order to adjust the size of the projected/collapsed chains. It has been tested with a collection of small to large programs including real applications (e.g., an interpreter, a compiler, an XSLT processor, etc.) producing good results, as summarized in Table 5.2. All the information related to the experiments, the source code of the tool, the benchmarks, and other materials can be found at:

`http://www.dsic.upv.es/~jsilva/DDJ/experiments.html`

Benchmark	ET nodes	Prj./Col.	Prj./Col. nodes	Bal. time	Quest.	Quest. bal.	%
NumReader	12 nodes	0/0	0/0 nodes	0 msec	6,46	6,46	0,00 %
Orderings	72 nodes	2/14	5/45 nodes	0 msec	11,47	8,89	22,46 %
Factoricer	62 nodes	7/0	17/0 nodes	0 msec	13,89	7,90	43,09 %
Sedgewick	41 nodes	3/8	7/24 nodes	0 msec	18,79	7,52	59,95 %
Clasifier	30 nodes	4/7	10/20 nodes	0 msec	15,52	6,48	58,21 %
LegendGame	93 nodes	12/20	28/40 nodes	0 msec	16,00	9,70	39,36 %
Cues	19 nodes	3/1	8/2 nodes	0 msec	10,40	8,20	21,15 %
Romanic	123 nodes	20/0	40/0 nodes	0 msec	25,06	16,63	33,66 %
FibRecursive	6724 nodes	19/1290	70/2593 nodes	344 msec	38,29	21,47	43,92 %
Risk	70 nodes	7/8	19/43 nodes	0 msec	30,69	10,28	66,50 %
FactTrans	198 nodes	5/0	12/0 nodes	0 msec	18,96	14,25	24,88 %
RndQuicksort	88 nodes	3/3	9/0 nodes	0 msec	12,88	10,40	19,20 %
BinaryArrays	132 nodes	7/0	18/0 nodes	0 msec	15,56	10,58	32,03 %
FibFactAna	380 nodes	3/29	9/58 nodes	0 msec	30,13	29,15	3,27 %
NewtonPol	46 nodes	1/3	2/40 nodes	0 msec	23,09	4,77	79,35 %
RegresionTest	18 nodes	1/0	3/0 nodes	0 msec	6,84	6,26	8,46 %
BoubleFibArrays	214 nodes	0/40	0/83 nodes	0 msec	12,42	12,01	3,33 %
ComplexNumbers	68 nodes	17/9	37/18 nodes	16 msec	20,62	10,20	50,53 %
StatsMeanFib	104 nodes	3/20	6/56 nodes	0 msec	12,33	11,00	10,81 %
Integral	25 nodes	0/2	0/22 nodes	0 msec	8,38	3,38	59,63 %
TestMath	51 nodes	1/2	2/5 nodes	0 msec	12,77	11,65	8,73 %
TestMath2	267 nodes	7/13	16/52 nodes	31 msec	66,47	58,33	12,24 %
Figures	116 nodes	8/3	16/6 nodes	0 msec	13,78	12,17	11,66 %
FactCalc	105 nodes	3/11	8/32 nodes	0 msec	19,81	12,64	36,19 %
SpaceLimits	127 nodes	38/0	76/0 nodes	0 msec	40,85	29,16	28,61 %
Argparser	129 nodes	31/9	70/37 nodes	16 msec	20,78	12,71	38,85 %
Cglib	1216 nodes	67/39	166/84 nodes	620 msec	80,41	65,01	19,15 %
Javassist	1357 nodes	10/8	28/24 nodes	4.745 msec	79,52	77,50	2,54 %
Kxml2	1172 nodes	260/21	695/42 nodes	452 msec	79,61	28,21	64,56 %
HTMLcleaner	6047 nodes	394/90	1001/223 nodes	8.266 msec	169,49	138,85	18,08 %
Jtstcase	4151 nodes	299/27	776/54 nodes	1.328 msec	85,05	80,52	5,32 %
Average	750,23 nodes	179,43/406,16	466,45/828,81 nodes	2.818,25 msec	85,97	68,21	29,86 %

Table 5.2: Benchmark results for the Tree Balancing technique

Each benchmark has been evaluated assuming that the bug could be at any node. This means that each row of the table is the average of a number of experiments. For instance, `cglib` was tested 1.216 times (i.e., the experiment was repeated choosing a different node as buggy, and all nodes were tried). For each benchmark, the `ET nodes` column shows the size of the evaluated ET; the `Prj./Col.` column shows the number of projection/collapse nodes inserted into the EF; the `Prj./Col. nodes` column shows the number of nodes that were projected and collapsed by the debugger; the `Bal. time` column shows the time needed by the debugger to balance the whole EF; the `Quest.` column shows the average number of questions using Top-Down done by the debugger before finding the bug in the original ET; the `Quest. bal.` column shows the average number of questions using Top-Down done by the debugger before finding the bug in the balanced ET; finally, the `(%)` column shows the improvement achieved with the balancing technique. Clearly, the balancing technique has an important impact in the reduction of questions with a mean reduction of 30 % using Top-Down.

In summary, our debugger produces the EF associated with any method execution specified by the user (by default `main`) and transforms it by collapsing and projecting nodes using Algorithm 9. Finally, it is explored with standard search strategies to find a bug. If we observe again Algorithms 6, 7, and 8, a moment of thought should convince the reader that their cost is linear with respect to the branching factor of the EF. In contrast, the cost of Algorithm 9 is quadratic with respect to the branching factor of the EF. On the practical side, our experiments reveal that the average cost of a single collapse (considering the 1.675 collapses) is 0,77 msec, and the average cost of a single projection (considering the 1.235 projections) is 17,32 msec. Finally, the average cost for balancing an EF is 2.818,25 msec.

Our algorithm is very conservative because it only collapses or projects nodes that belong to a chain. Our first experiments showed that if we do not apply any restrictions in the use of chains, the technique produces EFs that are much more balanced. Repeating the experiments in this way (considering all 23.257 experiments) produced a query reduction of 42 %. However, this reduction comes with a cost: the complexity of the questions may be increased. Therefore, we only apply the transformations when the question produced is not complicated (i.e., when it is generated by a chain). This has produced good results. In the case that the question of a collapse/projection node was still hard to answer, our tool gives the possibility of answering "I don't know", thus skipping the current question and continuing the debugging process with the other questions (e.g., with the children). This means that, if the user is able to find the bug with the standard ET, they will also be able with the balanced EF. That is, the introduction of projection nodes is conservative and cannot cause the debugging session to stop.

5.2.6 Related work

We are not aware of other approaches for balancing the structure of the ET. However, besides our approach, there exist other transformations devoted to reducing the size of the ET, and thus the number of questions performed. Our implementation allows for balancing an already generated ET, and it also allows for automatically generating the balanced ET. This can be done by collapsing or projecting chains during their generation. However, conceptually, our technique is a post-ET generation transformation.

The most similar approach is the Tree Compression technique introduced by Dave and Chitil [24]. This approach is also a conservative approach that transforms an ET into an equivalent (smaller) ET where the same bugs can be detected. The objective of this technique is essentially different: it tries to reduce the size of the ET by removing redundant nodes, and it is only applicable to recursive calls. A similar approach to tree compression is Declarative Source Debugging [17], which, instead of modifying the tree, it implements an algorithm to prevent the debugger from selecting questions related to nodes generated by recursive calls.

Another approach which is related to ours was presented in [78], where a transformation for list comprehensions of functional programs was introduced. In this case, it is a source code (rather than an ET) transformation to translate list comprehensions into equivalent functions that implement the iteration. The produced ET can be further transformed to remove the internal nodes of the ET reducing the size of the final ET as in the tree compression technique. Both techniques are orthogonal to the balancing of the ET, thus they both can be applied before balancing.

5.2.7 Proofs of technical results

Lemma 5.2.16 (Buggy method) *Given an EF $T = (N, E)$, and a buggy node $n \in N$ with $n = (b, m, e)$, then m contains a bug.*

Proof. Because n is buggy, then the method execution (b, m, e) is wrong, thus $(b, m, e) \notin \mathcal{I}$ and $\mathcal{I} \not\models (b, m, e)$. Moreover, by Definition 4.2.1, we have a child of n for each call to a method performed from the definition of m. But we know by Definition 4.2.3 that for all child n' of n, $n' \in \mathcal{I}$ or $\mathcal{I} \models n'$ Hence, m contains a bug. □

Proposition 5.2.17 *Let T be an EF with a wrong root. Then T contains a buggy node.*

Proof. We prove the claim by induction on the size of T.

(Base case) T only contains one node b. Then b is buggy, because it is wrong and it has no children.

(Induction hypothesis) T contains k nodes and at least one of them is buggy.

(Inductive case) T contains $k+1$ nodes. In this case we have a tree of k nodes that, by the induction hypothesis, does contain a buggy node b plus one extra node n. If n is not the child of b, then b is buggy. If n is the child of b, then either n is correct, and thus b is buggy; or n is wrong and hence, it is buggy because it has no children.

□

Lemma 5.2.18 (Soundness of projections and collapses) *Given a collapse or projection node $n = (b, m_1; \ldots; m_k, e)$ in an EF. If n is buggy, then it is associated with a buggy method.*

Proof. We have two possibilities:

- n **is a collapse node:** In this case n has not children and $m_1 = m_2 = \ldots = m_k = m$. Because n is wrong, then $(b, m; \ldots; m, e) \notin \mathcal{I}$ by Definition 5.2.3; therefore, trivially, m is buggy.
- n **is a projection node:** This case is impossible because a projection node cannot be buggy. The reason is that if all the children of n are correct, then n is correct by Definition 5.2.3 using the inference rule Tr, and hence n is not buggy. Otherwise, at least one child is wrong, but then, n cannot be buggy by Definition 4.2.3.

□

Theorem 5.2.12 (Completeness and soundness of EFs) *Given an EF with a wrong root, it contains a buggy node which is associated with a buggy method.*

Proof. The first point is proved by Proposition 5.2.17, while the second one is proved by Lemmas 5.2.16 and 5.2.18. □

Theorem 5.2.13 (Chain Collapse Correctness) *Let $T = (N, E)$ and $T' = (N', E')$ be two EFs, being the root of T wrong, and let $C \subset N$ be a chain such that all nodes in the chain are leaves and they have the same associated method. Given $T' =$ `collapseChain(T, C)`,*

1. *T' contains a buggy node.*
2. *Every buggy node in T' is associated with a buggy method.*

Proof. For the first item, only leaf nodes can be collapsed, therefore, the root node could only be collapsed if it is the only node of T. However, even in this case we have that $\nexists n \in N$ such that $(n \to r) \in E$ being r the root of T. Therefore, according to Definition 5.2.9, r is not a chain and thus it cannot be collapsed. Hence, the root of T' is the same as the root of T, and thus T' contains a buggy node by Proposition 5.2.17.

Now, we prove that any buggy node of T' is associated with a buggy method. Let $p \in N$ be the parent node of the chain C, and let $w \in N'$ be the collapse node of C. We consider three cases:

- $b \in N', p \neq b \neq w$ **is buggy:** In this case the collapse does not influence the buggy node b and thus the claim follows by Lemma 5.2.16.

- **p is buggy in T':** This case is trivial, because p is wrong and w is correct by Definition 4.2.3. Therefore, the new node w can be inferred with the rule Tr and thus the method in p is wrong according to Lemma 5.2.16.

  This case is particularly interesting because it reveals a phenomenon: node p is not changed by the transformation and thus it belongs to both trees T and T'. However, it could be possible that p is not buggy in T but it is buggy in T'. This happens because w has somehow hidden some error in the chain—some wrong intermediate result that was visible in the chain is now hidden because only the initial and final contexts are shown—, revealing a new bug located in p.

- **w is buggy in T':** Then, either the result or the final context of w are wrong. Hence, since both the result and the final context are produced by the nodes in C, we know that at least one node $c \in C$ is also wrong. Because c is a leaf and it is wrong, then it is a buggy node in T and it is associated with a buggy method m by Lemma 5.2.16. According to Algorithm 6, w is associated with a method execution $(s, m_1 \ldots m \ldots m_k, e)$, and thus it is associated with a buggy method.

□

Theorem 5.2.14 (Chain Projection Correctness) *Let $T = (N, E)$ and $T' = (N', E')$ be two EFs, and let $C \subset N$ be a chain such that $T' =$* `projectChain(T, C)`*.*

1. *All buggy nodes in T are also buggy nodes in T'.*

2. *Every buggy node in T' is associated with a buggy method.*

Proof. Let $p \in N$ be the parent node of the chain C, and let $w \in N'$ be the projection node of C. We consider an arbitrary buggy node $b \in N$ and show that it is also buggy in T'. Four cases are possible:

- **$p \neq b \neq w$ and $b \neq c \in C$:** In this case the projection does not influence either the buggy node b or its children and thus b is also buggy in T'.

- **$b = p$:** This means that b is the parent of the chain C. If it is buggy, then by Definition 4.2.3 all nodes in C are correct. Then, as shown in the proof of Lemma 5.2.18, w is correct. Hence, b is also buggy in T'.

  In the general case, p will be buggy in T, and also in T' ($\forall c \in C$, c will be correct; and thus w is also correct). However, it could be possible that p is wrong in T, two nodes $c_1, c_2 \in C$ were wrong, and their combined (wrong) effects produced a correct result. In that case, both errors would be hidden in the projection node (but of course they would remain in c_1 and c_2). As a result, a new buggy node (p) not present in T would appear in T'.

- **$b = w$:** This case is impossible as shown in the proof of Lemma 5.2.18.

- **$b = c \in C$:** This means that c is wrong and all its children correct. Since the children of c have not been modified by `projectChain`, c was buggy in T and it is also buggy in T'.

In all cases, the buggy node is associated with a buggy method by Lemmas 5.2.16 and 5.2.18.

□

5.2.8 Case of study: Mergesort

In this section we show a real application of our algorithm using our implementation. In particular, we show the result of balancing an EF produced from a Mergesort program. The following Mergesort algorithm version was initially extracted from Wikipedia (`https://en.wikipedia.org/wiki/Merge_sort`) and then modified to include one bug in the `merge` function: it does not update positions appropriately.

The `main` method declares an object of the `mergesort` class, initializes the `x` variable with the list `{3,1,2}`, and sorts `x` with `OrderMerge`. Finally, the result is stored in the `y` variable, and it is printed:

```java
import java.util.Random;

public class mergesort {
    public static void main(String[] args) {
        mergesort m = new mergesort();
        int[] x = { 3, 1, 2 };
        int[] y = m.OrderMerge(x);

        for (int i = 0; i < y.length; i++)
            System.out.println(y[i]);
    }
```

The `OrderMerge` method implements the mergesort algorithm: First, it splits the list given as argument; then, it sorts each fragment and merges them together to obtain the final result:

```java
    public int[] OrderMerge(int[] L) {
        int n = L.length;
        if (n > 1) {
            int m = (int) (Math.ceil(n / 2.0));
            int[] L1 = new int[m];
            int[] L2 = new int[n - m];

            for (int i = 0; i < m; i++)
                L1[i] = L[i];
            for (int i = m; i < n; i++)
                L2[i - m] = L[i];
            L = merge(OrderMerge(L1), OrderMerge(L2));
        }
        return L;
    }
```

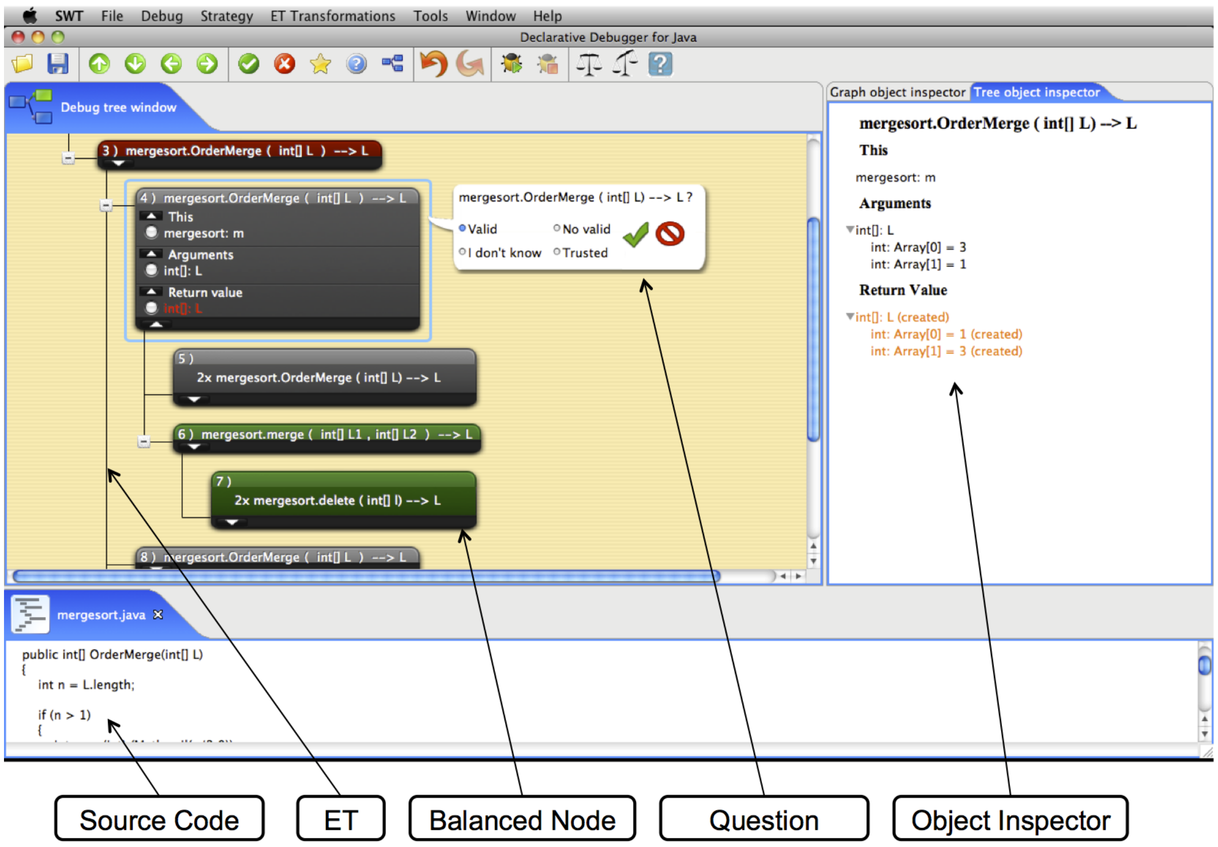
The `merge` method introduces in a new list the elements of the (sorted) list received as arguments in an ordered fashion, and the list is returned as the result. However, we have introduced an error: when the element in the first array is smaller than the one in the second array, it is introduced in the new array, *but the position is not updated.* Thus, the next introduced element will overwrite its value:

```java
    public int[] merge(int[] L1, int[] L2) {
        int[] L = new int[L1.length + L2.length];
        int i = 0;

        while ((L1.length != 0) && (L2.length != 0)) {
            if (L1[0] < L2[0]) {
                L[i] = L1[0];
                // Bug!! Didn't update the position
                // Correct: L[i++] = L1[0];
                L1 = delete(L1);
                if (L1.length == 0)
                    while (L2.length != 0) {
                        L[i++] = L2[0];
                        L2 = delete(L2);
                    }
            } else {
                L[i++] = L2[0];
                L2 = delete(L2);
                if (L2.length == 0)
                    while (L1.length != 0) {
                        L[i++] = L1[0];
                        L1 = delete(L1);
                    }
            }
        }
        return L;
    }
```

Finally, the `delete` method removes the first element of the list received as argument:

```
    public int[] delete(int[] l) {
        int[] L = new int[l.length - 1];

        for (int i = 1; i < l.length; i++)
            L[i - 1] = l[i];
        return L;
    }
}
```

If we execute the main method of the class, we would observe that the initial list, `{3,1,2}`, is sorted to `{2,3,0}`. Since the expected result was `{1,2,3}`, there must be a bug in the code. At this point, we can use a conventional debugger and place breakpoints at some parts of the code to try to figure out whether the variables are updated correctly at those points. Then, add new breakpoints, and so on.



```
3 ) mergesort.OrderMerge ( int[] L ) --> L
4 ) mergesort.OrderMerge ( int[] L ) --> L
    This
    mergesort: m
    Arguments
    int[]: L
    Return value
    int[]: L
5 )
    2x mergesort.OrderMerge ( int[] L) --> L
6 ) mergesort.merge ( int[] L1 , int[] L2 ) --> L
7 )
    2x mergesort.delete ( int[] l) --> L
8 ) mergesort.OrderMerge ( int[] L ) --> L
```

mergesort.OrderMerge (int[] L) --> L ?
[✓ Valid] No valid
I don't know Trusted

```
mergesort.OrderMerge ( int[] L) --> L
This
mergesort: m
Arguments
int[]: L
    int: Array[0] = 3
    int: Array[1] = 1
Return Value
int[]: L (created)
    int: Array[0] = 1 (created)
    int: Array[1] = 3 (created)
```

```
public int[] OrderMerge(int[] L)
{
   int n = L.length;

   if (n > 1)
   {
```

Figure 5.7: AD session of Mergesort with DDJ

An alternative is to use our debugger. When we load *mergesort.java* in the debugger we see the information shown in Figure 5.7. In the main panel we can observe a part of the EF associated with Mergesort. The debugger uses colours to distinguish between those nodes that are marked as wrong (in red), and those nodes that are marked as correct (in green). The validity of the grey nodes is unknown, and thus they are suspicious of being buggy. Of course, the user can inspect any node at any point, and they can also direct the debugging session manually as it happens with breakpoints. But in general, the user allows the debugger to automatically direct the search for the bug, because it always selects the node that better divides the suspicious area.

In the figure, the debugger has automatically selected a node related with the `OrderMerge` method, and it has prompted the question associated with this node:

```
mergesort.OrderMerge({3,1}) = {1,3}?
```

Clearly the answer to the question is `Valid` because the elements of the array have been correctly ordered. Note also that it is not even necessary to look at the implementation of the method to answer the question; i.e., to answer we focus on the objective of the method (we only have to know that this method should order an array), and we do not need to know the operational details (it does not matter how it was implemented).

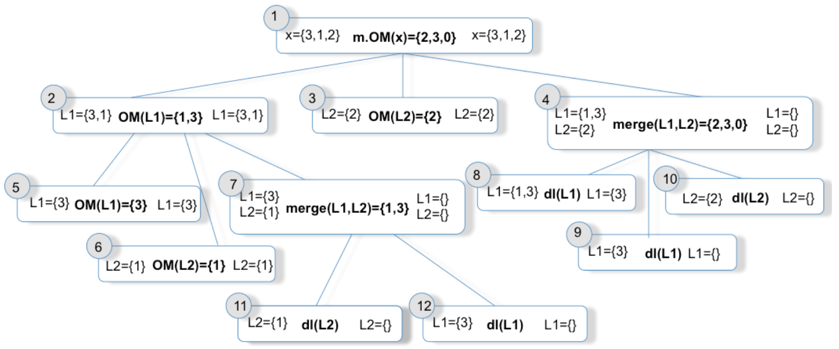


Figure 5.8: EF associated with Mergesort

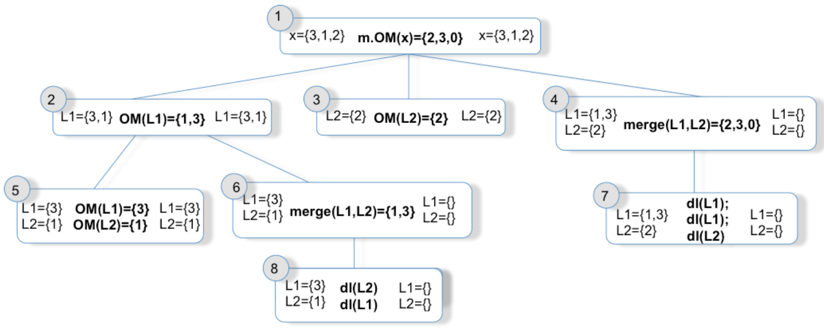

1
x={3,1,2} m.OM(x)={2,3,0} x={3,1,2}
2
L1={3,1} OM(L1)={1,3} L1={3,1}
3
L2={2} OM(L2)={2} L2={2}
4
L1={1,3} L2={2} merge(L1,L2)={2,3,0} L1={} L2={}
5
L1={3} L2={1} OM(L1)={3} OM(L2)={1} L1={3} L2={1}
6
L1={3} L2={1} merge(L1,L2)={1,3} L1={} L2={}
7
L1={1,3} L2={2} dl(L1); dl(L1); dl(L2) L1={} L2={}
8
L1={3} L2={1} dl(L2) dl(L1) L1={} L2={}

Figure 5.9: Balanced EF associated with Mergesort

The information associated to each node is displayed in the panel on the right (the object inspector). It is similar to the "variable watch view" used in traditional debuggers: It allows for inspecting all objects in the scope of this method. Note that the information is classified so that we can separately see the arguments and the result; and creation, deletion or changes in any object are highlighted with colours.

The EF associated with Mergesort is displayed in Figure 5.8, where `OM` abbreviates `OrderMerge`. A standard algorithmic debugger would traverse this tree as follows:

```
Starting Debugging Session...
(1) NO  (2) YES  (3) YES  (4) NO  (8) YES  (9) YES  (10) YES
Bug located in method: merge(int[] L1, int[] L2) of the Mergesort class
```

Fortunately, using our balancing algorithm, the debugger can transform the EF in Figure 5.8 to produce the EF of Figure 5.9, where Nodes (5) and (6); (8), (9) and (10); and, (11) and (12), have been collapsed. The tool automatically detected that they form chains and collapsed them to form a single question. For instance, the new collapse node (7) in Figure 5.9 contains the following question: *"Having the L1 = { 1, 3 } and L2 = { 2 } lists, if we delete two elements from the L1 list, and one element from the L2 list; should we obtain the L1 = { } and L2 = { } lists?"*

Although larger trees would not fit in these pages, we can already see the pattern of collapsing for this example: merging two lists of sizes $n1$ and $n2$, that in general produces $n1 + n2 + 1$ nodes in the EF (one node for `merge` and $n1 + n2$ for `delete`), only produces two nodes in the balanced EF.

With the balanced EF, the traversal is:

```
Starting Debugging Session...
(1) NO  (2) YES  (3) YES  (4) NO  (7) YES
Bug located in method: merge(int[] L1, int[] L2) of the Mergesort class
```

5.3 Loops to recursion

5.3.1 Introduction

Iteration and recursion are two different ways to reach the same objective. In some paradigms, such as the functional or logic, iteration does not even exist. In other paradigms, e.g., the imperative or the object-oriented paradigm, the programmer can decide which of them to use. However, they are not totally equivalent, and sometimes it is desirable to use recursion, while other times iteration is preferable. In particular, one of the most important differences is the performance achieved by both of them. In general, compilers have produced more efficient code for iteration, and this is the reason why several transformations from recursion to iteration exist (see, e.g., [42, 62, 71]). Recursion in contrast is known to be more intuitive, reusable and debuggable. Another advantage of recursion shows up in presence of hierarchized memories. In fact, other researchers have obtained both theoretical and experimental results showing significant performance benefits of recursive algorithms on both uniprocessor hierarchies and on shared-memory systems [104]. In particular, Gustavson and Elmroth [39, 26] have demonstrated significant performance benefits from recursive versions of Cholesky and QR factorization, and Gaussian elimination with pivoting.

Recently, a new technique for algorithmic debugging [56] revealed that transforming all iterative loops into recursive methods before starting the debugging session can improve the interaction between the debugger and the user, and it can also reduce the granularity of the located errors. In particular, algorithmic debuggers only report buggy methods. Thus, a bug inside a loop is reported as a bug in the whole method that contains the loop, which is sometimes too imprecise. Transforming a loop into a recursive method allows the debugger to identify the recursive method (and thus the loop) as buggy. Hence, we wanted to implement this transformation and integrate it in the *Declarative Debugger for Java* (DDJ), but, surprisingly, we did not find any available transformation from iterative loops into recursive methods for Java (or for any other object-oriented language). Therefore, we had to implement it by ourselves and decided to automatize and generalize the transformation to make it publicly available. To the best of our knowledge this is the first transformation for all types of iterative loops. Moreover, our transformation handles exceptions and accepts the use of any number of *break* and *continue* statements (with or without labels).

One important property of our transformation [54] is that it always produces tail recursive methods [20]. This means that they can be compiled into efficient code because the compiler only needs to keep two activation records in the stack to execute the whole loop [40, 3]. Another important property is that each iteration is always represented with one recursive call. This means that a loop that performs 100 iterations is transformed into a recursive method that performs 100 recursive calls. This equivalence between iterations and recursive calls is very important for some applications such as debugging, and it produces code that is more maintainable.

The objective of this section is twofold. On the one hand, it is a description of a transformation explained in such a way that one can study the transformation of a specific construct (e.g., exceptions) without the need to see how other constructs, such as the *return* statement, are transformed. This decomposition of the transformation into independent parts can be very useful for academic purposes. In particular, this section describes the transformation step by step using different sections to explain the treatment of advanced features such as exception handling and the use of labels. Because we are not aware of any other publicly available description, some parts can help students and beginner programmers to completely understand and exercise the relation between iteration and recursion, while other more advanced parts can be useful for the implementors of the transformation. On the other hand, the proposed transformation has been implemented as a publicly available library. To the best of our knowledge, this is the first automatic transformation for an object-oriented language that is complete (i.e., it accepts the whole language).

Example 5.3.1 *Transforming loops into recursion is necessary in many situations (e.g., compilation to functional or logic languages, algorithmic debugging, program understanding, memory hierarchies optimization, etc.). However, the transformation of a loop into an equivalent recursive method is not trivial at all in the general case. For this reason, there exist previous ad-hoc implementations that cannot accept the whole language, or that are even buggy. For instance, the transformation proposed in [30] does not accept exceptions and it crashes in situations like the following:*

```
for (int i = 0; i < 10; i++)
    for (int j = 0; j < 10; j++)
        break;
```

due to a bug in the implementation.

Consider the Java code in Algorithm 10. It is not particularly complicated, but it shows some of the difficulties that can appear during a transformation.

Algorithm 10 Iterative loop with exceptions

```
 1: public int example(int x) throws IOException {
 2:     loop1:
 3:     while (x < 10) {
 4:         try {
 5:             x = 42 / x;
 6:         } catch (Exception e) { break loop1; }
 7:         loop2:
 8:         for (int i = 1; i < x; i++)
 9:             if (x % i > 0);
10:                 throw new Exception1();
11:             else continue loop1;
12:     }
13:     return x;
14: }
```

*This algorithm contains two nested loops (*while *and* for*). Therefore, it would be normally translated into recursion using three methods, one for the original method* example*, one for the outer loop* loop1*, and one for the inner loop* loop2*. However, the use of exceptions and statements such as* break *and* continue *poses restrictions on the implementation of these methods. For instance, observe in line 11 that the control can pass from one loop to the other due to the use of the* loop1 *label. This forces the programmer to implement some mechanism to record the values of all variables shared by both loops and pass the control from one loop to the other when this point is reached. Note also that this change in the control could affect several levels (e.g., if a* break *is used in a deeper loop). In addition, the use of exceptions imposes additional difficulties. Observe for instance that the inner loop throws an exception* Exception1 *in line 10. This exception could inherit from* IOException *and thus it should be captured in the* loop2 *method and passed in some way to the* loop1 *method that in turn should decide if it catches the exception or passes it to the* example *method that would throw it. To the best of our knowledge, this example cannot be translated into recursion by any of the already existing transformations.*

In the rest of this section we describe our transformation for all types of loops in Java (i.e., *while/-do/for/foreach*). The transformation of each particular type of loop is explained with an example. We start with an illustrative example that provides the reader with a general view of how the transformation works.

Algorithm 11 Sqrt (iterative version)

```
public double sqrt(double x) {
    if (x < 0)
        return Double.NaN;
    double b = x;
    while (Math.abs(b * b - x) > 1e-12)
        b = ((x / b) + b) / 2;
    return b;
}
```

Example 5.3.2 *Consider the Java code in Algorithm 11. It computes the square root of the input argument.*
This algorithm implements a while*-loop where each iteration obtains a more accurate approximation of the square root of the* x *variable. The transformed code is depicted in Algorithm 12 that implements the same functionality but replacing the* while*-loop with a new recursive method sqrt_loop.*

Algorithm 12 Sqrt (recursive version)

```
 1: public double sqrt(double x) {
 2:     if (x < 0)
 3:         return Double.NaN;
 4:     double b = x;
 5:     if (Math.abs(b * b - x) > 1e-12)
 6:         b = this.sqrt_loop(x, b);
 7:     return b;
 8: }
 9: private double sqrt_loop(double x, double b) {
10:     b = ((x / b) + b) / 2;
11:     if (Math.abs(b * b - x) > 1e-12)
12:         return this.sqrt_loop(x, b);
13:     return b;
14: }
```

Essentially, the transformation performs two steps:

1. Substitute a new code for the original loop (lines 5-6 in Algorithm 12).

2. Create a new recursive method (lines 9-14 in Algorithm 12).

In Algorithm 12, the new code in the *sqrt* method includes a call (line 6) to the recursive method *sqrt_loop* that implements the loop (lines 9-14). This new recursive method contains the body of the original loop (line 10). Therefore, each time the method is invoked, an iteration of the loop is performed. The rest of the code added during the transformation (lines 5, 11-13) is the code needed to simulate the same effects of a *while*-loop. Therefore, this is the only code that we should change to adapt the transformation to the other types of loops (*do/for/foreach*).

The rest of this section is organized as follows. In Section 5.3.2 we introduce our transformations for each particular type of loop and provide detailed examples and explanations. In Section 5.3.3 we extend our algorithms with a special treatment for *break* and *continue* statements. Section 5.3.4 presents the transformation in presence of exceptions and errors (*try*, *catch*, *throw* and the hierarchy of objects that inherit from *Throwable*). Section 5.3.5 presents the transformation in presence of recursion and the *return* and *goto* statements. Section 5.3.6 presents the implementation of the technique, some optimizations that can be applied to the general transformation, and an empirical evaluation with a benchmark suite. Section 5.3.7 discusses some related approaches. Finally, Section 5.3.8 provides a proof of correctness of the transformation.

5.3.2 Transforming loops into recursive methods

Our program transformations are summarized in Table 5.3. This table has a different row for each type of loop. For each loop, we have two columns. One for the iterative version of the loop, and one for the transformed recursive version. Observe that the code is presented in an abstract way, so that it is formed by a parameterized skeleton of the code that can be instantiated with any particular loop of each type.

In the recursive version, the code inside the ellipses is code inserted by the programmer (it comes from the iterative version). The rest of the code is automatically generated by the transformation. Here, `result` and `loop` are fresh names (not present in the iterative version) for a variable and a method, respectively; `type` is a data type that corresponds to the data type declared by the user (it is associated with a variable already declared in the iterative version). The code inside the squares has the following meaning:

$\boxed{1}$ contains the sequence formed by all the variables declared in `Code1` (and in `ini` in *for*-loops) that are used in `Code2` and `cond` (and in `upd` in *for*-loops).

$\boxed{1'}$ contains the previous sequence but including types (because it is used as the parameters of the method, and the previous sequence is used as the arguments of the call to the method).

$\boxed{2}$ contains for each object in the `result` array (which contains the same variables as $\boxed{1}$ and $\boxed{1'}$), a casting of the object to assign the corresponding type. For instance, if the array contains two variables [`x,y`] whose types are respectively `double` and `int`; then $\boxed{2}$ contains:
```
x = (Double) result[0];
y = (Integer) result[1];
```

Observe that, even though these steps are based on Java, the same steps (with small modifications) can be used to transform loops in many other imperative or object-oriented languages. The code in Table 5.3 is generic. In some specific cases, this code can be optimized. For instance, observe that the recursive method always returns an array of objects (`return new Object[] {...}`) with all variables that changed in the loop. This array is unnecessary and inefficient if the recursive method only needs to return one variable (or if it does not need to return any variable). Therefore, the creation of the array should be replaced with a single variable or null (i.e., `return null`). In the rest of the section, we always apply optimizations when possible, so that the code does not perform any unnecessary operations. This allows us to present a generic transformation as the one in Table 5.3, and also to provide specific efficient transformations for each type of loop. The optimizations are not needed to understand the transformation, but they should be considered when implementing it. Therefore, we will explain the optimizations in detail in the implementation section. In the rest of this section we explain the transformation of all types of loop. The four types of loops (*while/do/for/foreach*) present in the Java language behave nearly in the same way. Therefore, the modifications needed to transform each type of loop into a recursive method are very similar. We start by describing the transformation for *while*-loops, and then we describe the variations needed to adapt the transformation for *do/for/foreach*-loops.

	Iterative version	Recursive version	
		Caller	Recursive method

While-loop	Code 1
	while (cond) {
	Code 2
	}
	Code 3
		Code 1
		if (cond) {
		Object[] result = loop($\boxed{1}$);
		$\boxed{2}$
		}
		Code 3
			private Object[] loop($\boxed{1^*}$) {
			Code 2
			if (cond)
			return loop($\boxed{1}$);
			return new Object[] { $\boxed{1}$ };
			}

Do-loop	Code 1
	do {
	Code 2
	} while (cond);
	Code 3
		Code 1
		{
		Object[] result = loop($\boxed{1}$);
		$\boxed{2}$
		}
		Code 3
			private Object[] loop($\boxed{1^*}$) {
			Code 2
			if (cond)
			return loop($\boxed{1}$);
			return new Object[] { $\boxed{1}$ };
			}

For-loop	Code 1
	for (ini; cond; upd) {
	Code 2
	}
	Code 3
		Code 1
		{
		ini;
		if (cond) {
		Object[] result = loop($\boxed{1}$);
		$\boxed{2}$
		}
		}
		Code 3
			private Object[] loop($\boxed{1^*}$) {
			Code 2
			upd;
			if (cond)
			return loop($\boxed{1}$);
			return new Object[] { $\boxed{1}$ };
			}

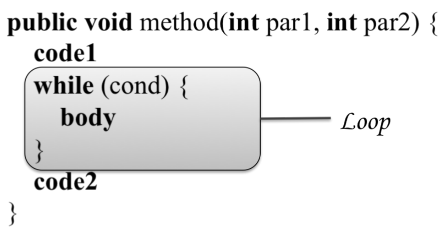
Foreach-loop (Array version)	Code 1
	for (type elem : elems) {
	Code 2
	}
	Code 3
		Code 1
		if (0 < elems.length) {
		Object[] result = loop($\boxed{1}$, elems, 0);
		$\boxed{2}$
		}
		Code 3
			private Object[] loop($\boxed{1^*}$,
			type[] elems, int index) {
			type elem = elems[index];
			Code 2
			index++;
			if (index < elems.length)
			return loop($\boxed{1}$, elems, index);
			return new Object[] { $\boxed{1}$ };
			}

Foreach-loop (Iterable version)	Code 1
	for (type elem : elems) {
	Code 2
	}
	Code 3
		Code 1
		{
		Iterator<type> iter = elems.iterator();
		if (iter.hasNext()) {
		Object[] result = loop($\boxed{1}$, iter);
		$\boxed{2}$
		}
		}
		Code 3
			private Object[] loop($\boxed{1^*}$,
			Iterator<type> iter) {
			type elem = iter.next();
			Code 2
			if (iter.hasNext())
			return loop($\boxed{1}$, iter);
			return new Object[] { $\boxed{1}$ };
			}

Table 5.3: Taxonomy of the transformations from loops to recursion

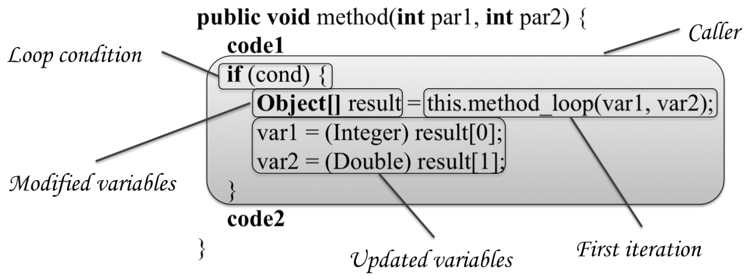
Transformation of while-loops

In Table 5.4 we show a general overview of the steps needed to transform a Java iterative *while*-loop into an equivalent recursive method. Each step is described in the following.



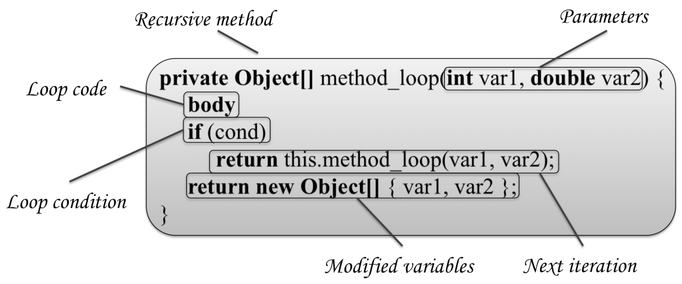
```
public void method(int par1, int par2) {
  code1
  while (cond) {
    body
  }
  code2
}
```

(a) Original method



```
public void method(int par1, int par2) {
  code1
  if (cond) {
    Object[] result = this.method_loop(var1, var2);
    var1 = (Integer) result[0];
    var2 = (Double) result[1];
  }
  code2
}
```

(b) Transformed method



```
private Object[] method_loop(int var1, double var2) {
  body
  if (cond)
    return this.method_loop(var1, var2);
  return new Object[] { var1, var2 };
}
```

(c) Recursive method

Figure 5.10: Transformation from a *while*-loop template to recursion

Step	Correspondence with Figure 5.10
	Figure 5.10(b)
1) Substitute the loop with a call to the recursive method	Caller
1.1) If the loop condition is satisfied	Loop condition
1.1.1) Perform the first iteration	First iteration
1.2) Catch the variables modified during the recursion	Modified variables
1.3) Update the modified variables	Updated variables
	Figure 5.10(c)
2) Create the recursive method	Recursive method
2.1) Define the parameters of the method	Parameters
2.2) Define the code of the recursive method	
2.2.1) Include the code of the original loop	Loop code
2.2.2) If the loop condition is satisfied	Loop condition
2.2.2.1) Perform the next iteration	Next iteration
2.2.3) Otherwise return the modified variables	Modified variables

Table 5.4: Steps for the transformation of the *while*-loop to recursion

Substitute the loop with a call to the recursive method. The first step is to remove the original loop and substitute it with a call to the new recursive method. We can see this substitution in Figure 5.10(b). Observe that some parts of the transformation have been labelled to ease later references to the code. The tasks performed during the substitution are explained in the following:

- **Perform the first iteration**
  In the *while*-loop, first of all we check whether the *loop condition* holds. If it does not hold, then the loop is not executed. Otherwise, the *first iteration* is performed by calling the recursive method with the variables used inside the loop as arguments of the method call. Hence, we need an analysis to know what variables are used inside the loop. The recursive method is in charge of

executing as many iterations of the loop as needed.

- **Catch the variables modified during the recursion**
  The variables modified during the recursion cannot be automatically updated in Java because all parameters are passed by value. Therefore, if we modify an argument inside a method we are only modifying a copy of the original variable. This also happens with objects. Hence, in order to output those *modified variables* that are needed outside the loop, we use an array of objects. Because the *modified variables* can be of any data type[6], we use an array of objects of class **Object**.

  In presence of call-by-reference, this step should be omitted.

- **Update the modified variables**
  After the execution of the loop, the *modified variables* are returned inside an **Object** array. Each variable in this array must be cast into its respective type before being assigned to the corresponding variable declared before the loop.

  In presence of call-by-reference, this step should be omitted.

Create the recursive method. Once we have substituted the loop, we create a new method that implements the loop in a recursive way. This *recursive method* is shown in Figure 5.10(c).

The code of the *recursive method* is explained in the following:

- **Define the parameters of the method**
  There are variables declared inside a method but declared outside the loop and used in this loop. When the loop is transformed into a *recursive method*, these variables are not accessible from inside the *recursive method.* Therefore, they must be passed as arguments in the calls to it. Hence, the parameters of the *recursive method* are the intersection between the variables declared before the loop and the variables used inside it.

- **Define the code of the recursive method**
  Each iteration of the original iterative loop is emulated with a call to the new recursive method. Therefore, in the code of the *recursive method* we have to execute the current iteration and control whether the *next iteration* must be executed or not.

  - **Include the code of the original loop**
    When the *recursive method* is invoked it means that we want to execute one iteration of the loop. Therefore, we place the *original code* of the loop at the beginning of the *recursive method.* This code is supposed to update the variables that control the *loop condition.* Otherwise, the original loop is in fact an infinite loop and the recursive method created will be invoked infinitely.

  - **Perform the next iteration**
    Once the iteration is executed, we check the *loop condition* again to know whether another iteration must still be executed. In such a case, we perform the *next iteration* with the same arguments. Note that the values of the arguments can be modified during the execution of the iteration, therefore, each iteration has different argument values, but the names and the number of arguments remain always the same.

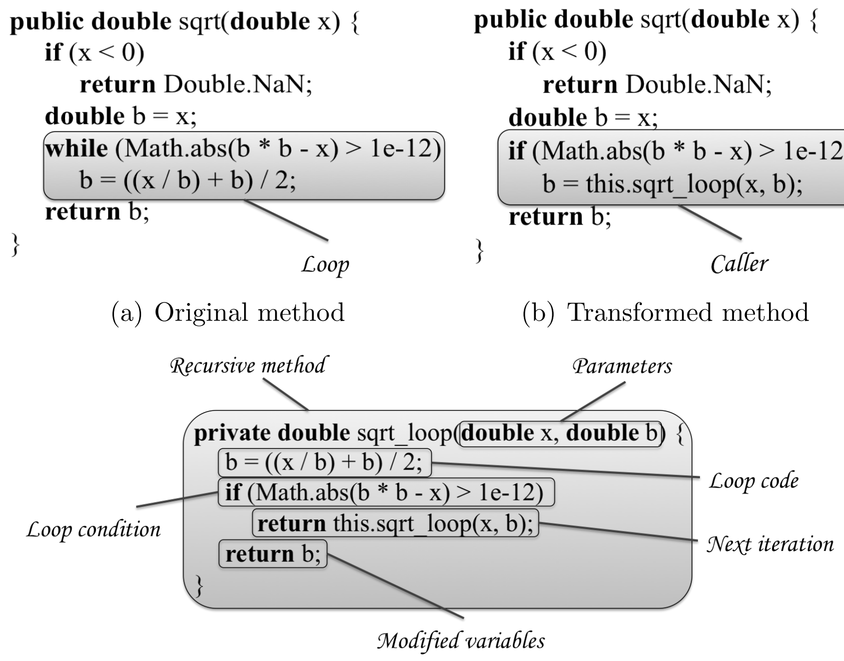
  - **Otherwise return the modified variables**
    If the loop condition does not hold, the loop ends and thus we must finish the sequence of

[6]In the case that the returned values are primitive types, then they are naturally encapsulated by the compiler in their associated primitive wrapper classes.

recursive method calls and return to the original method in order to continue executing the rest of the code. Because the arguments have been updated in each recursive call, at this point we have the last values of the variables involved in the loop. Hence, these variables must be returned in order to update them in the original method. Observe that these variables are passed from iteration to iteration during the execution of the recursive method until it is finally returned to the recursive method caller.

In presence of call-by-reference, this step should be omitted.

Figure 5.11 shows an example of a transformation of a *while*-loop.



```
public double sqrt(double x) {
    if (x < 0)
        return Double.NaN;
    double b = x;
    while (Math.abs(b * b - x) > 1e-12)
        b = ((x / b) + b) / 2;
    return b;
}
```

(a) Original method

```
public double sqrt(double x) {
    if (x < 0)
        return Double.NaN;
    double b = x;
    if (Math.abs(b * b - x) > 1e-12)
        b = this.sqrt_loop(x, b);
    return b;
}
```

(b) Transformed method

```
private double sqrt_loop(double x, double b) {
    b = ((x / b) + b) / 2;
    if (Math.abs(b * b - x) > 1e-12)
        return this.sqrt_loop(x, b);
    return b;
}
```

(c) Recursive method

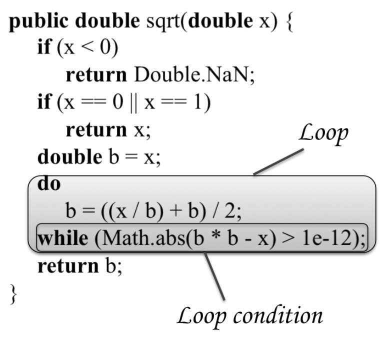
Figure 5.11: Transformation from the *while*-loop to recursion

Transformation of do-loops

do-loops behave exactly in the same way as *while*-loops except in one detail: The first iteration of the *do*-loop is always performed. In Figure 5.12(a) we can see an example of a *do*-loop. This code obtains the square root value of the x variable as the code in Algorithm 11. The difference is that, if the x variable is either 0 or 1, then the method directly returns the x variable, otherwise the loop is performed in order to calculate the square root. In order to transform the *do*-loop into a recursive method, we can follow the same steps used in Table 5.4 with only one change: in step 1.1 the loop condition is not evaluated; instead, we only need to add a new code block to ensure that those variables created during the transformation are not available outside the transformed code.

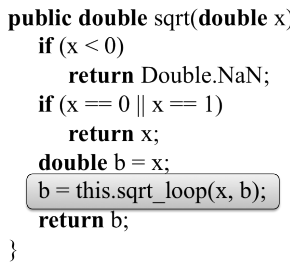
Figure 5.12(b) and 5.12(c) illustrates the only change needed to transform the *do*-loop into a recursive method. Observe that in this example there is no need to introduce a new block, because the transformed code does not create new variables, but in the general case the block could be needed.

- **Add a new code block**
  Observe in Table 5.3, in the `Caller` column, that, contrarily to *while*-loops, *do*-loops need to introduce a new *block* (i.e., a new scope). The reason is that there could exist variables with the



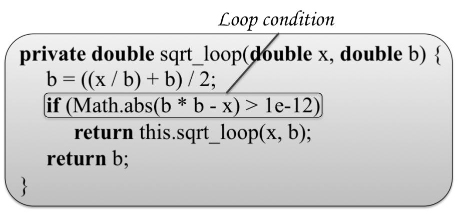
```
public double sqrt(double x) {
    if (x < 0)
        return Double.NaN;
    if (x == 0 || x == 1)
        return x;
    double b = x;
    do
        b = ((x / b) + b) / 2;
    while (Math.abs(b * b - x) > 1e-12);
    return b;
}
```

(a) Do loop



```
public double sqrt(double x) {
    if (x < 0)
        return Double.NaN;
    if (x == 0 || x == 1)
        return x;
    double b = x;
    b = this.sqrt_loop(x, b);
    return b;
}
```

(b) Recursive method caller



```
private double sqrt_loop(double x, double b) {
    b = ((x / b) + b) / 2;
    if (Math.abs(b * b - x) > 1e-12)
        return this.sqrt_loop(x, b);
    return b;
}
```

(c) Recursive method

Figure 5.12: Transformation from the do-loop to recursion

same name as the variables created during the transformation (e.g., *result*). Hence, the new block avoids variable clashes and limits the scope of the variables created by the transformation.

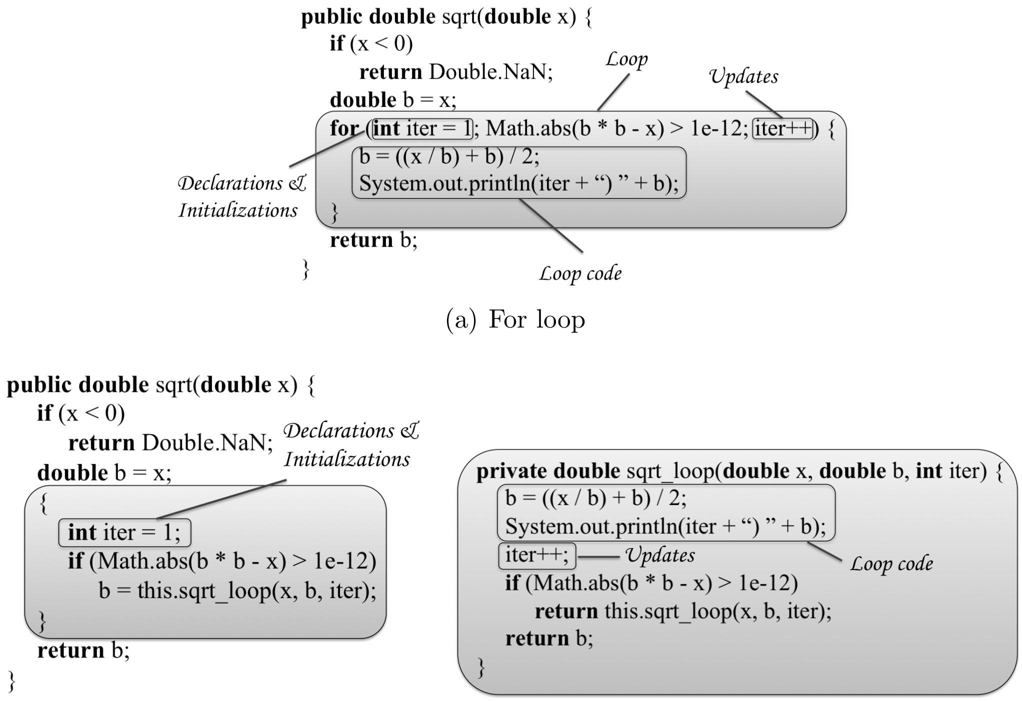
Transformation of for-loops

One of the most frequently used loops in Java is the *for*-loop. This loop behaves exactly in the same way as the *while*-loop except in one detail: *for*-loops provide the programmer with a mechanism to declare, initialize and update variables that will be accessible inside the loop.

In Figure 5.13(a) we can see an example of a *for*-loop. This code obtains the square root value of the x variable exactly as the code in Algorithm 11, but it also prints the approximation obtained in every iteration. We can see in Figure 5.13(b) and 5.13(c) the additional changes needed to transform the *for*-loop into a recursive method.

As shown in Figure 5.13, in order to transform the *for*-loop into a recursive method, we can follow the same steps used in Table 5.4, but we have to make three changes:

- **Add a new code block**
  Exactly in the same way and with the same purpose as in *do*-loops.
- **Add the declarations/initializations at the beginning of the block**
  In the original method, those variables created during the *declaration* and *initialization* of the loop are only available inside it (and not in the code that follows the loop). We must ensure that these variables maintain the same scope in the transformed code. This can be easily achieved with the new *block*. In the transformed code, these variables are declared and initialized at the beginning of the new *block*, and they are passed as arguments to the recursive method in every iteration to make them accessible inside it.
- **Add the updates between the loop code and the loop condition**
  In *for*-loops there exists the possibility of executing code between iterations. This code is usually a



```
public double sqrt(double x) {
    if (x < 0)
        return Double.NaN;
    double b = x;
    for (int iter = 1; Math.abs(b * b - x) > 1e-12; iter++) {
        b = ((x / b) + b) / 2;
        System.out.println(iter + “) ” + b);
    }
    return b;
}
```

Loop
Updates
Declarations & Initializations
Loop code

(a) For loop

```
public double sqrt(double x) {
    if (x < 0)
        return Double.NaN;
    double b = x;
    {
        int iter = 1;
        if (Math.abs(b * b - x) > 1e-12)
            b = this.sqrt_loop(x, b, iter);
    }
    return b;
}
```

Declarations & Initializations

```
private double sqrt_loop(double x, double b, int iter) {
    b = ((x / b) + b) / 2;
    System.out.println(iter + “) ” + b);
    iter++;
    if (Math.abs(b * b - x) > 1e-12)
        return this.sqrt_loop(x, b, iter);
    return b;
}
```

(b) Recursive method caller (c) Recursive method

Figure 5.13: Transformation from the for-loop to recursion

collection of *updates* of the variables declared at the beginning of the loop (e.g., in Figure 5.13(a) this code is `iter++`). However, this code could be formed by a series of expressions separated by commas that could include method invocations, assignments, etc. Because this *update* code is always executed before the condition of the loop, it must be placed in the recursive method between the *loop code* and the *loop condition*.

Transformation of foreach-loops

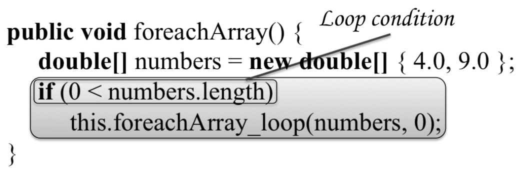
foreach-loops are especially useful to traverse collections of elements. In particular, this type of loops traverses a given collection and it executes a block of code for each element. The transformation of a *foreach*-loop into a recursive method is different depending on the type of collection that is traversed. In Java we can use *foreach*-loops either with *arrays* or *iterable* objects. We explain each transformation separately.

***foreach*-loop used to traverse arrays.** An array is a composite data structure where elements have been sequentialized, and thus, they can be traversed linearly. We can see an example of a *foreach*-loop that traverses an array in Algorithm 13.

Algorithm 13 *foreach*-loop that traverses an array (iterative version)

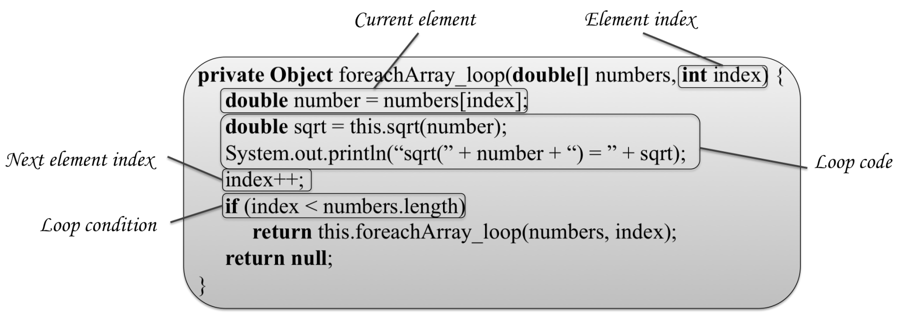
```
public void foreachArray() {
    double[] numbers = new double[] { 4.0, 9.0 };
    for (double number : numbers) {
        double sqrt = this.sqrt(number);
        System.out.println(“sqrt(” + number + “) = ” + sqrt);
    }
}
```

This code computes and prints the square root of all elements in the [4.0, 9.0] array. Each individual square root is computed with Algorithm 11. The *foreach*-loop traverses the array sequentially starting from *numbers[0]* until the last element in the array. The transformation of this loop into an equivalent recursive method is very similar to the transformation of a *for*-loop. However, there are differences. For instance, *foreach*-loops lack a counter. This can be observed in Figure 5.14 that implements a recursive method equivalent to the loop in Algorithm 13.



```
public void foreachArray() {
    double[] numbers = new double[] { 4.0, 9.0 };
    if (0 < numbers.length)
        this.foreachArray_loop(numbers, 0);
}
```

(a) Recursive method caller



```
private Object foreachArray_loop(double[] numbers, int index) {
    double number = numbers[index];
    double sqrt = this.sqrt(number);
    System.out.println("sqrt(" + number + ") = " + sqrt);
    index++;
    if (index < numbers.length)
        return this.foreachArray_loop(numbers, index);
    return null;
}
```

(b) Recursive method

Figure 5.14: Transformation from the *foreach*-loop to recursion (Array version)

In Figure 5.14 we can see the symmetry with respect to the *for*-loop transformation. The only difference is the creation of a fresh variable that is passed as argument in the recursive method calls (in the example this variable is called *index*). This variable is used for:

- **Controlling whether there are more elements to be treated**
  A *foreach*-loop is only executed if the array contains elements. Therefore we need a *loop condition* in the recursive method caller and another in the recursive method to know when there are no more elements in the array and thus finish the traversal. The later is controlled with a variable (*index* in the example) acting as a counter.

- **Obtaining the next element to be treated**
  During each iteration of the *foreach*-loop a variable called *number* is instantiated with one element of the array (line 3 of Algorithm 13). In the transformation this behaviour is emulated by declaring and initializing this variable at the beginning of the recursive method. It is initialized to the corresponding element of the array by using the *index* variable.

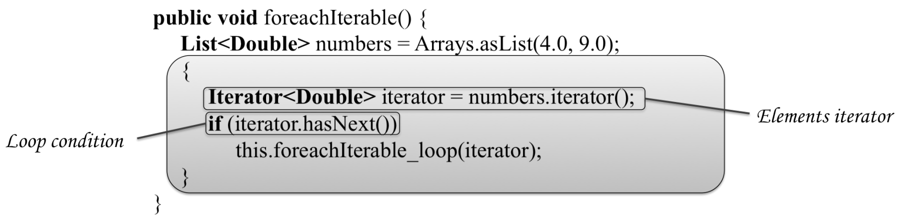
***foreach*-loop used to traverse *iterable* objects.** A *foreach*-loop can be used to traverse objects that implement the *Iterable* interface. Algorithm 14 shows an example of a *foreach*-loop using one of these objects.

This code behaves exactly in the same way as Algorithm 13 but using an *iterable* object instead of an array (*numbers* is an iterable object because it is an instance of the *List* class that in turn implements the *Iterable* interface). The *Iterable* interface only has one method, called *iterator*, that returns an object that implements the *Iterator* interface. With regard to the *Iterator* interface, it forces the programmer to implement the *next*, *hasNext* and *remove* methods; and these methods allow the programmer to freely

Algorithm 14 *foreach*-loop used to traverse an iterable object (iterative version)

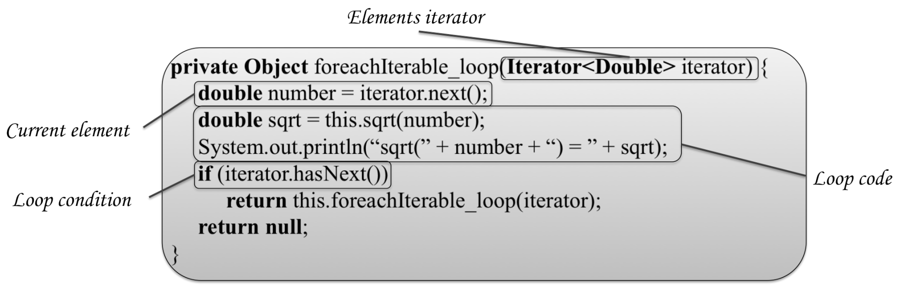
```
public void foreachIterable() {
    List<Double> numbers = Arrays.asList(4.0, 9.0);
    for (double number : numbers) {
        double sqrt = this.sqrt(number);
        System.out.println("sqrt(" + number + ") = " + sqrt);
    }
}
```

implement how the collection is traversed (e.g., the order, whether repetitions are taken into account or not, etc.). Therefore, the transformed code should use these methods to traverse the collection. We can see in Figure 5.15 a recursive method equivalent to Algorithm 14.



```
public void foreachIterable() {
   List<Double> numbers = Arrays.asList(4.0, 9.0);
   {
      Iterator<Double> iterator = numbers.iterator();
      if (iterator.hasNext())
         this.foreachIterable_loop(iterator);
   }
}
```

(a) Recursive method caller



```
private Object foreachIterable_loop(Iterator<Double> iterator){
   double number = iterator.next();
   double sqrt = this.sqrt(number);
   System.out.println("sqrt(" + number + ") = " + sqrt);
   if (iterator.hasNext())
      return this.foreachIterable_loop(iterator);
   return null;
}
```

(b) Recursive method

Figure 5.15: Transformation from the foreach-loop to recursion (Iterable version)

Observe that the transformed code in Figure 5.15 is very similar to the one in Figure 5.14. The only difference is the use of an *iterator* variable (instead of an integer variable) that controls the element of the collection to be treated. Note that the *next* method of the *iterator* variable makes it possible to know what the next element to be treated is, and the *hasNext* method tells us whether there still exist more elements to be processed.

5.3.3 Treatment of *break* and *continue* statements

The *break* and *continue* control statements can change the normal control flow of a loop. These commands can also be used in combination with a parameter that specifies the specific loop that should be continued or broken. We can illustrate their use with a Java program extracted from `http://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html` and shown in Algorithm 15. This program takes two strings and decides whether the second is a substring of the first.

In the example, the ***continue*** *test* statement (line 11) is used to continue the search in the next substring of the *searchMe* variable. Since it contains the *test* label, it affects the *for*-loop (labelled with "*test*") instead of the *while*-loop (as it would happen if *continue* was unlabelled).

Algorithm 15 Use of the statement *continue* with a label

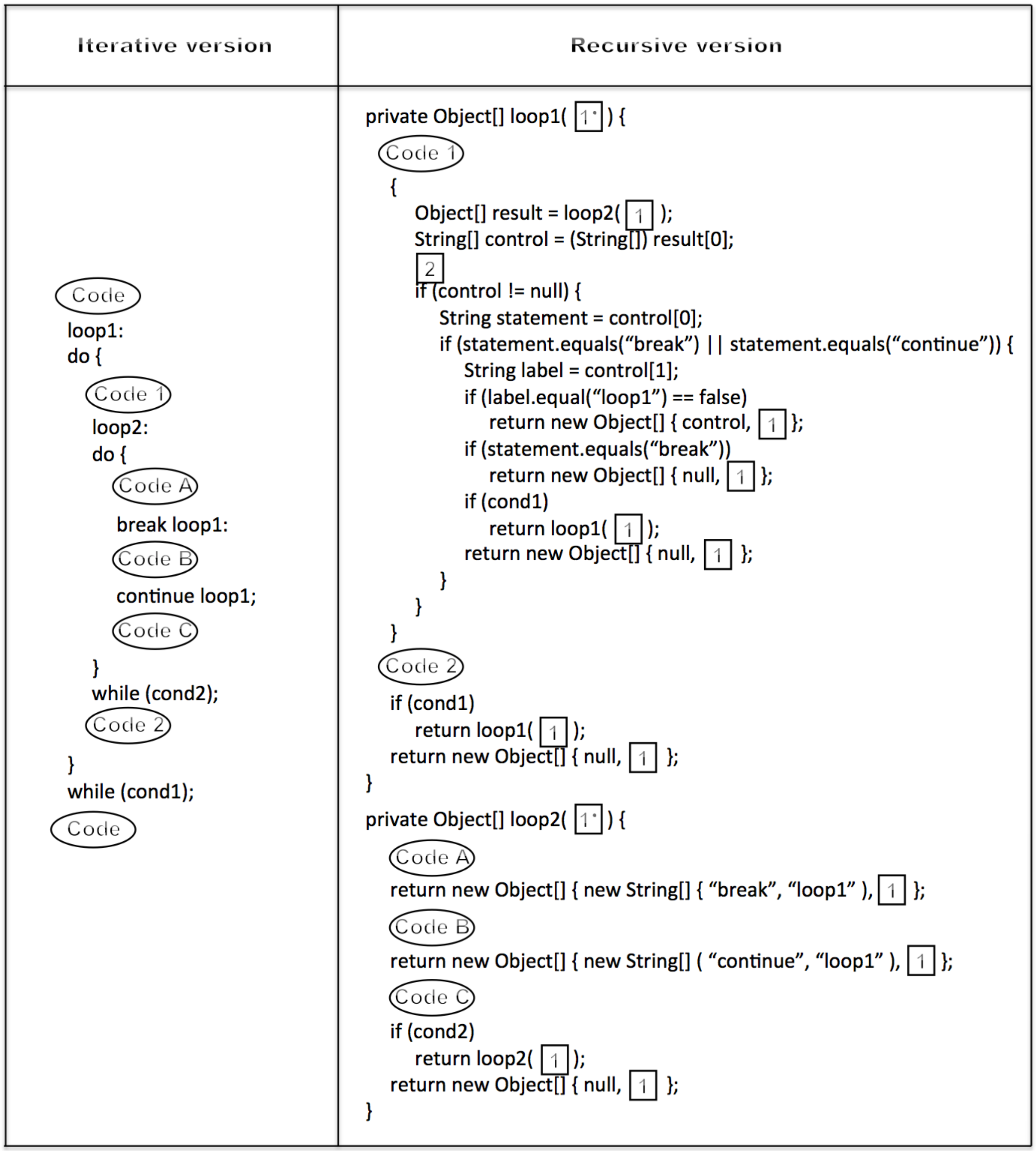
```
public void substr() {
    String searchMe = "Look for a substring in me";
    String substring = "sub";
    int max = searchMe.length() - substring.length();
    boolean foundIt = false;
    test:
    for (int i = 0; i <= max; i++) {
        int n = substring.length(), j = i, k = 0;
        while (n-- != 0)
            if (searchMe.charAt(j++) != substring.charAt(k++))
                continue test;
        foundIt = true;
        break test;
    }
    System.out.println(foundIt ? "Found it" : "Didn't find it");
}
```

Table 5.5 presents a schema that summarizes the additional treatments that are necessary in presence of *break* and *continue* statements. We have not found any public report for any language that describes how to transform these statements. This table is general enough as to be able to work with any number of combined (possibly nested) loops and *break/continue* statements with or without labels. The table includes the case where *break/continue* are included inside a nested loop (*loop2*). Note however that there could be other loops in the different codes inside the ellipses. In such a case, the treatment would be extended exactly in the same way as it is described in the table for the case of two nested loops. Note also that *loop1* and *loop2* could be any type of loop (*while/do/for/foreach*), and thus the transformation should proceed in each type as stated in Table 5.3. Essentially, the right column in Table 5.5 performs the following steps:

- In the output of all recursive methods (i.e., in the fresh variable *result*), we always return in the first position (*result[0]*) an array of strings. This array of strings (referred to with the *control* fresh variable) contains two flags to indicate to the outer recursive methods (representing the outer loops) whether a *break* or *continue* statement that must be processed has been executed in the inner recursive methods (representing the inner loops). In the case that none of them was executed, or if they do not have a label associated, then *control* is *null*. The flags of *control* are the following:
    - *statement*: indicates whether a *break* or *continue* has been executed.
    - *label*: indicates the label associated with *statement*.
- After a call to a recursive method, the transformed code examines the output and depending on the values in the flags it behaves differently:
    - if *control* is *null*, then no *break/continue* statement has been executed or it has already been processed by the inner recursive methods. Hence, the execution proceeds normally. Otherwise, if the statement inside the control variable is *break* or *continue*, then
    - if the label of the current loop is different from the returned *label*, this means that a *break/-continue* statement has been executed but it affects another outer loop. Therefore, we exit this method (without further executing anything else) and continue with the outer method.

In this case, we have to output *control* to indicate to the external methods that one of them should be broken or continued.

- if the label of the current loop is equal to the returned *label*, and *statement* is equal to *break*, we exit this method and set *control* to *null* so that external methods proceed normally.
- if the label of the current loop is equal to the returned *label*, and *statement* is equal to *continue*, this method ends its current iteration and continues with the next by performing a recursive call.



Iterative version

```
Code
loop1:
do {
  Code 1
  loop2:
  do {
    Code A
    break loop1;
    Code B
    continue loop1;
    Code C
  }
  while (cond2);
  Code 2
}
while (cond1);
Code
```

Recursive version

```
private Object[] loop1( [1*] ) {
  Code 1
  {
    Object[] result = loop2( [1] );
    String[] control = (String[]) result[0];
    [2]
    if (control != null) {
      String statement = control[0];
      if (statement.equals("break") || statement.equals("continue")) {
        String label = control[1];
        if (label.equal("loop1") == false)
          return new Object[] { control, [1] };
        if (statement.equals("break"))
          return new Object[] { null, [1] };
        if (cond1)
          return loop1( [1] );
        return new Object[] { null, [1] };
      }
    }
  }
  Code 2
  if (cond1)
    return loop1( [1] );
  return new Object[] { null, [1] };
}
private Object[] loop2( [1*] ) {
  Code A
  return new Object[] { new String[] { "break", "loop1" ), [1] };
  Code B
  return new Object[] { new String[] ( "continue", "loop1" ), [1] };
  Code C
  if (cond2)
    return loop2( [1] );
  return new Object[] { null, [1] };
}
```

Table 5.5: Schema showing the transformation of *break* and *continue* statements to recursion

We can see the behaviour of this table in a concrete example. Algorithm 16 implements a recursive method equivalent to Algorithm 15. This example is especially interesting because it combines the transformation of two nested loops (a *for*-loop and a *while*-loop) together with the use of a *break* and a *continue* statements that use a label of the same loop where they are declared (the *break* statement) and a label of an outer loop (the *continue* statement).

Algorithm 16 Transformation of a *continue* with label

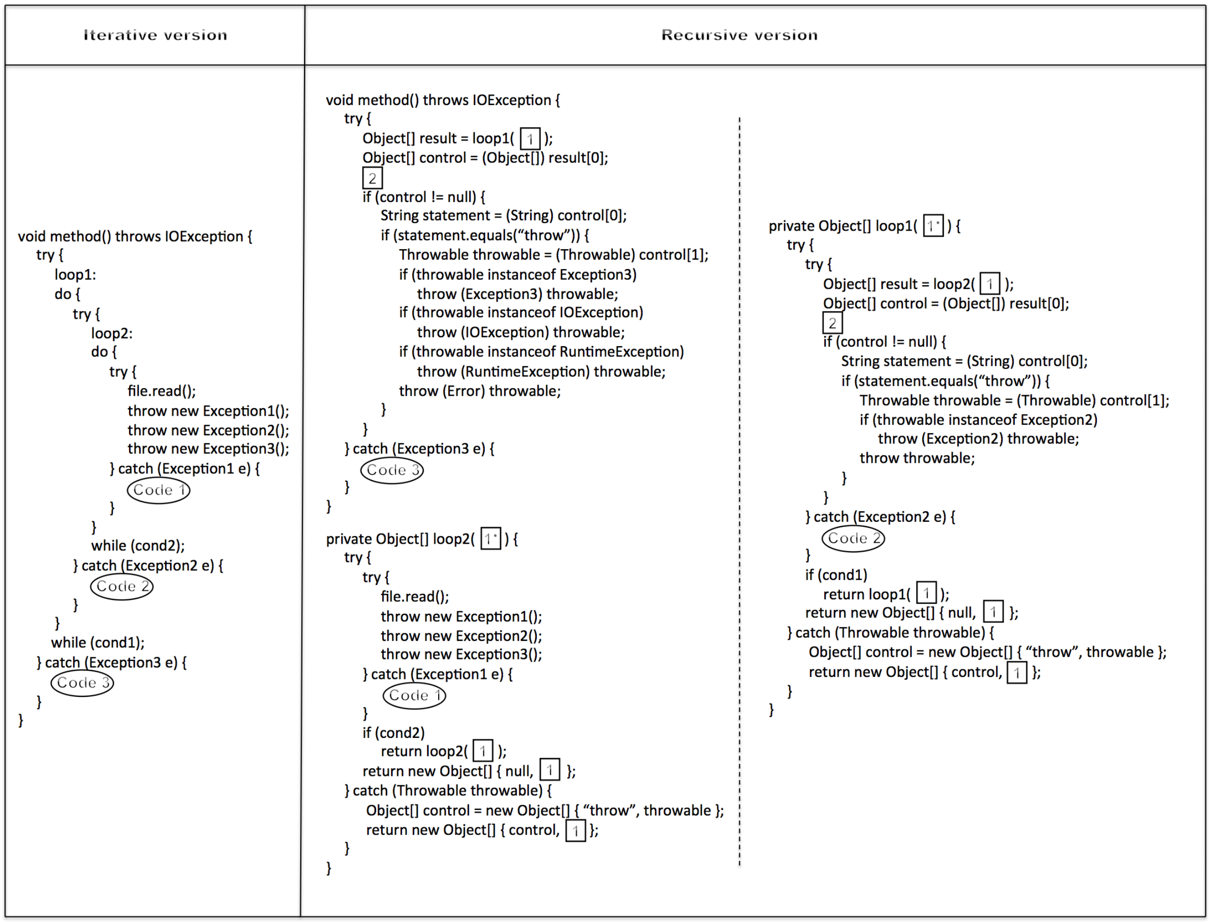
```
 1: public void substr() {
 2:     String searchMe = "Look for a substring in me";
 3:     String substring = "sub";
 4:     int max = searchMe.length() - substring.length();
 5:     boolean foundIt = false;
 6:     {
 7:         int i = 0;
 8:         if (i <= max)
 9:             foundIt = substr_test(searchMe, substring, max, foundIt, i);
10:     }
11:     System.out.println(foundIt ? "Found it" : "Didn't find it");
12: }
13: private boolean substr_test(String searchMe, String substring, int max, boolean foundIt, int i) {
14:     int n = substring.length(), j = i, k = 0;
15:     if (n-- != 0) {
16:         String[] control = substr_test_loop(searchMe, substring, n, j, k);
17:         if (control != null) {
18:             i++;
19:             if (i <= max)
20:                 return substr_test(searchMe, substring, max, foundIt, i);
21:             return foundIt;
22:         }
23:     }
24:     foundIt = true;
25:     return foundIt;
26: }
27: private String[] substr_test_loop(String searchMe, String substring, int n, int j, int k) {
28:     if (searchMe.charAt(j++) != substring.charAt(k++))
29:         return new String[] { "continue", "test" };
30:     if (n-- != 0)
31:         return substr_test_loop(searchMe, substring, n, j, k);
32:     return null;
33: }
```

In this example, *loop1* and *loop2* correspond to *for*- and *while*-loops respectively. We can observe the analogy with the treatment described in Table 5.5. In particular, in Algorithm 15, inside the *for*-loop, the *break* statement (line 13) refers to this loop (due to the *test* label). Hence, in Algorithm 16 we substitute it with a return statement that just ends the execution of the loop (line 25). Contrarily, inside the *while*-loop, the *continue* statement (line 11) refers to the outer loop (due again to the *test* label). Therefore, we substitute it with a return statement where we set the control flags to their corresponding values (line 29).

5.3.4 Handling exceptions

Exceptions interrupt the normal execution of a program. They can be thrown due to two reasons: explicitly thrown by the programmer (using the *throw* command) or implicitly thrown by executing an instruction (e.g., a division by zero, opening a file that does not exist, etc.). When an exception is thrown, there must exist a part of the code that catches this exception and handles it by executing some subroutines. Therefore, all methods in a program (and the methods generated by our transformation in particular) must be labelled with which exceptions are caught by them, and thus the other exceptions would be thrown to the caller of the method.

Table 5.6 presents a schema that summarizes the additional treatment necessary to handle exceptions. In the left column we have a method with two nested loops and four different exceptions that are thrown inside the inner loop. *Exception1* is caught inside the inner loop (*loop2*); *Exception2* is caught inside the outer loop (*loop1*); *Exception3* is caught outside both loops but inside the method; and *IOException*, which may be thrown due to the *file.read()* statement, is thrown out of the method.



Iterative version

```
void method() throws IOException {
    try {
        loop1:
        do {
            try {
                loop2:
                do {
                    try {
                        file.read();
                        throw new Exception1();
                        throw new Exception2();
                        throw new Exception3();
                    } catch (Exception1 e) {
                        (Code 1)
                    }
                }
                while (cond2);
            } catch (Exception2 e) {
                (Code 2)
            }
        }
        while (cond1);
    } catch (Exception3 e) {
        (Code 3)
    }
}
```

Recursive version

```
void method() throws IOException {
    try {
        Object[] result = loop1( [1] );
        Object[] control = (Object[]) result[0];
        [2]
        if (control != null) {
            String statement = (String) control[0];
            if (statement.equals("throw")) {
                Throwable throwable = (Throwable) control[1];
                if (throwable instanceof Exception3)
                    throw (Exception3) throwable;
                if (throwable instanceof IOException)
                    throw (IOException) throwable;
                if (throwable instanceof RuntimeException)
                    throw (RuntimeException) throwable;
                throw (Error) throwable;
            }
        }
    } catch (Exception3 e) {
        (Code 3)
    }
}

private Object[] loop2( [1•] ) {
    try {
        try {
            file.read();
            throw new Exception1();
            throw new Exception2();
            throw new Exception3();
        } catch (Exception1 e) {
            (Code 1)
        }
        if (cond2)
            return loop2( [1] );
        return new Object[] { null, [1] };
    } catch (Throwable throwable) {
        Object[] control = new Object[] { "throw", throwable };
        return new Object[] { control, [1] };
    }
}
```

```
private Object[] loop1( [1•] ) {
    try {
        try {
            Object[] result = loop2( [1] );
            Object[] control = (Object[]) result[0];
            [2]
            if (control != null) {
                String statement = (String) control[0];
                if (statement.equals("throw")) {
                    Throwable throwable = (Throwable) control[1];
                    if (throwable instanceof Exception2)
                        throw (Exception2) throwable;
                    throw throwable;
                }
            }
        } catch (Exception2 e) {
            (Code 2)
        }
        if (cond1)
            return loop1( [1] );
        return new Object[] { null, [1] };
    } catch (Throwable throwable) {
        Object[] control = new Object[] { "throw", throwable };
        return new Object[] { control, [1] };
    }
}
```

Table 5.6: Transformation of exceptions to recursion

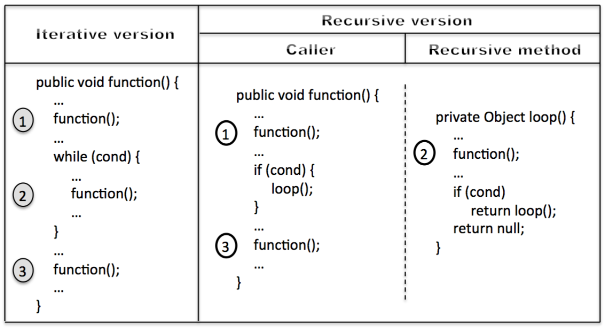
The right column shows the transformed code. The transformation uses the same mechanism used for *break* and *continue*. In each method that represents a loop, it uses the *result* array to output the exceptions that have occurred inside the method and must be caught by other method (i.e., the caller or any other previous one). Observe the transformed code for the inner loop (*loop2* method). This method catches all exceptions that were already caught by the original loop (i.e., *catch (Exception1 e)*). The other exceptions should be caught outside of the method (i.e., *Exception2*, *Exception3* and *IOException*). Therefore, the *loop2* method includes a *try-catch* that surrounds all statements. This *try-catch* catches any possible throwable exception because it catches a generic *Throwable* object. This is necessary to be able to catch all exceptions thrown by the programmer, and also those that can be thrown by the system. When an exception is caught, it is returned to the caller method inside an array called *control* with a *"throw"* flag to indicate that an exception that must be handled is being returned.

In the outer loop (*loop1* method), after the call to *loop2*, we check whether the *control* variable is *null*. This is the way to know if the method should proceed normally, or it should handle the exception. Whenever *control != null*, it checks whether the exception is *Exception2*. In such a case, the exception is thrown, caught, and handled by this method. Otherwise, it is thrown again to the caller of this method. This process is repeated inside each transformed method that represents a loop. Note in the example that the *IOException* exception is not handled by any method, and thus, it will be thrown by the initial method exactly as it happens in the original code.

There is a detail in the transformed *method* that is important to remark and explain. In Java there are two types of exceptions, named *checked* and *unchecked*. While *checked* exceptions must be always explicitly caught or thrown, *unchecked* exceptions are implicitly handled by the compiler (i.e., there is no need to include them in the *throws* statement). Moreover, all *unchecked* exceptions inherit from *RuntimeException* or *Error*. Therefore, in the case that (i) an exception reaches the transformed *method* (thus, it was not handled inside any loop), (ii) the exception is not handled by this method (it is not *Exception3*), and also, (iii) it does not appear in the *throws* section (it is not *IOException*), then this exception is necessarily an *unchecked* exception and must be thrown as such. To allow the compiler to catch an *unchecked* exception (e.g., *NullPointerException* in *file.read()* if *file* is *null*), the transformation must use a casting before throwing it (in the figure, *throw (RuntimeException) throwable* and *throw (Error) throwable*). This ensures that all checked and unchecked exceptions are caught.

5.3.5 Treatment of recursion and the `return` and `goto` statements

In this section we explain what happens when the loop contains recursive calls, and how to treat *return* and *goto* commands. We start explaining how to transform loops into recursive calls. Roughly, recursion does not need special treatment. Recursive calls are treated as any other statement in the loop, as it is shown in the programs in Table 5.7.



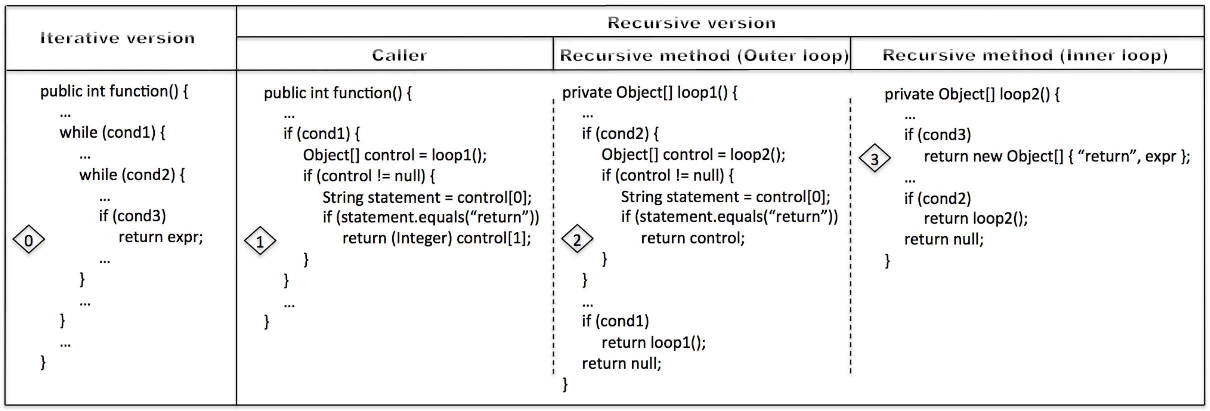
Iterative version	Recursive version	
	Caller	Recursive method
public void function() {		
...	public void function() {	
① function();	...	private Object loop() {
...	① function();	...
while (cond) {	...	② function();
...	if (cond) {	...
② function();	loop();	if (cond)
...	}	return loop();
}	...	return null;
...	③ function();	}
③ function();	...	
...	}	
}		

Table 5.7: Transformation of recursive methods to recursion

In the iterative version of the loop we have three recursive calls (numbered 1, 2 and 3). Recursive calls 1 and 3 are outside the loop. They both are executed only once before and after the loop respectively in the original code. Analogously, they are executed before and after the transformed code associated with the loop. Therefore, trivially, provided that the transformed code behaves as the loop does, then they do not affect the transformation and should remain unchanged. Contrarily, recursive call 2 is done inside the loop and, thus, it will be executed in every iteration of the loop. The transformed code places the recursive call inside the new generated recursive method (`loop`). Each activation of this method corresponds to one iteration of the original loop. Note that the recursive call to `function` is treated as any other statement. If it produces a change in the state of the loop (like, e.g., an assignment), it will affect the rest of the code in the same way in both the original and the transformed code.

The transformation of *return* statements is very similar to the one for *break* statements. Both statements force the control to stop the execution, leave the current loop, and continue the execution in another point of the code. We illustrate the transformation of *return* with Table 5.8.

The code in Table 5.8 shows a *return* inside a nested loop. Therefore, it forces the execution to leave both loops and exit the `function` method. Observe that the transformed code is completely analogous



Iterative version	Recursive version		
	Caller	Recursive method (Outer loop)	Recursive method (Inner loop)

public int function() {
  ...
  while (cond1) {
    ...
    while (cond2) {
      ...
      if (cond3)
◇0   return expr;
      ...
    }
    ...
  }
  ...
}
	public int function() {
	  ...
	  if (cond1) {
	    Object[] control = loop1();
	    if (control != null) {
	      String statement = control[0];
	      if (statement.equals("return"))
	◇1 return (Integer) control[1];
	    }
	  }
	  ...
	}
		private Object[] loop1() {
		  ...
		  if (cond2) {
		    Object[] control = loop2();
		    if (control != null) {
		      String statement = control[0];
		      if (statement.equals("return"))
		◇2 return control;
		    }
		  }
		  ...
		  if (cond1)
		    return loop1();
		  return null;
		}
			private Object[] loop2() {
			  ...
			  if (cond3)
			◇3 return new Object[] { "return", expr };
			  ...
			  if (cond2)
			    return loop2();
			  return null;
			}

Table 5.8: Transformation of *return* statements to recursion

to the one for *break* statements. The transformation converts the *return* in rhombus 0 to the *return* in rhombus 3. Observe that we use a flag with the `"return"` string to indicate to the outer loops that a *return* statement was executed. Then, all other methods check the flag and propagate the *return* statement (rhombus 2) until it reaches the original method (the caller). The caller removes the flag and returns the same value as in the original loop (rhombus 1).

Our transformation does not consider the use of *goto* statements. Note that, even though *goto* is a reserved word in Java, it is not used as a language construct, and thus, in practice, Java does not have *goto*.[7] If we had a *goto* statement inside a transformed loop, three situations would be possible: the *goto* statement points to a label inside (1) the same loop; (2) an inner loop; or (3) an outer loop. In the first case the *goto* statement can remain unchanged in the transformed code, and the transformation will perfectly work. Cases 2 and 3 are not handled by our transformation. However, in the third case, there can exist an equivalence between a *goto* statement and a labelled *break*/*continue* statement. In such cases the transformation of the *goto* statement would correspond to the transformation of a labelled *break*/*continue* statement.

5.3.6 Implementation as a Java library, optimizations and empirical evaluation

All the transformations described in this section have been implemented as a Java library called *loops2recursion*. This library consists of approximately 3,000 lines of Java code, and it is publicly available at:

`http://www.dsic.upv.es/~jsilva/loops2recursion/`

The library contains the implementation of all the individual algorithms needed to transform each type of Java loop, and it also contains generic code able to parse a whole Java file or project. Roughly, it parses the source code and it automatically detects and transforms all loops in the program.

The transformation can be used, e.g., as follows:

```
import loops2recursion;
   (...)
FileTransformer ft = new FileTransformer(sourceFile, targetFile);
```

[7]The *goto* command was included in the list of Java's reserved words because, if it were added to the language later on, existing code that used the word *goto* as an identifier (variable name, method name, etc.) would break. But because *goto* is a keyword, such code will not even compile in the present, and it remains possible to make it actually do something later on, without breaking existing code.

```
ft.openFile();
ft.showCode();
ft.loops2recursion();
ft.showCode();
ft.saveFile();
```

Observe that an object of the `FileTransformer` class is created to transform a file. Then, we can invoke several methods provided by the library to manipulate this object. In the example, we first open the source file (`openFile`), next we show in the console the code in the source file (`showCode`), then we perform the transformation (`loops2recursion`) and show the transformed code (`showCode`). Finally, we save the transformed code in the target file (`saveFile`).

Optimizations

The code in Table 5.3 is generic. In some specific cases, this code can be optimized. The following optimizations are not needed to understand the transformation, but they should be considered when implementing it (all of them are implemented by *loops2recursion*):

1. **Do not update unmodified variables.**
   If the loop does not modify a variable (e.g., it is only read), then do not return this variable in the recursive method. We can see an example in Algorithm 12 where the x variable is not modified during the execution of the *sqrt_loop* method and thus we do not return it (line 13).

2. **Variable uniformity.**
   If all variables returned in the recursive method are of the same type, then the returned array should be of that type (e.g., `int[]`). We can see an example in Algorithm 16 where the *substr_test_loop* method returns an array of strings (line 27).

3. **Avoid returning an array when possible.**
   When the transformed method only returns a single variable, then the array is unnecessary and only the variable should be returned. We can see an example in Algorithm 12 where we only return the b variable (line 13).

4. **Throw the exceptions (do not catch them) if there are no variables that should be updated.**
   In the generated code, exceptions are handled to update the variables when returning from the recursive method. If there are no variables to update, then it is not necessary to catch the exceptions.

5. **Avoid the external block when possible.**
   The transformation generates an external block to avoid that variables declared inside the generated code exist after it. If there are no declared variables, then the block is unnecessary. We can see an example in Algorithm 12 where the *result* variable is not necessary because we can directly assign the return value to the b variable (line 6).

6. **Remove unreachable code.**
   The transformation can generare unreachable code when the last statement of the loop is a *break*/*continue*/*return*/*throw* statement. In that case, the code generated after this sentence is unreachable and must be removed. We can see an example in Algorithm 16 where the *sqrt_test* method is the transformation of a *for*-loop. In this example the transformation would normally insert the code to execute the *next iteration* at the end of the method, but since the last statement of the loop is a break, then this code would be unreachable and it has been removed (line 25).

7. **Remove the treatment of exceptions and *break/continue/return* when it is unnecessary.**
   The code generated to treat exceptions should be generated only in the case that an exception can occur. The same happens with the code generated for *break/continue/return*. We can see an example in Algorithm 16 where the *substr_test* method directly performs the *next iteration* after checking that *control != null* (lines 18-21). The transformation can take this decision by analyzing the *substr_test_loop* method where *control != null* can only occur when the continue statement has been executed.

Empirical evaluation

We conducted several experiments to empirically evaluate both the transformation and the transformed code using the *loops2recursion* library. These experiments provide a precise and quantitative idea of the performance of the transformation.

For the evaluation, we selected a set of benchmarks from the Java Grande benchmark suite [9]. A Java Grande application is one that uses large amounts of processing, I/O, network bandwidth, or memory. They include not only applications in science and engineering but also, for example, corporate databases and financial simulations. Specific information about the benchmarks including the source code can be found at:

`https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/java-grande-benchmark-suite/`

We designed the experiment from a set of 14 Java Grande benchmarks. Firstly, we automatically transformed the *loops2recursion* library by self-application (i.e., all loops were translated to recursive methods). Then, we transformed the 14 benchmarks with the two versions of the library (the original, and the transformed one) and executed all of them several times to compare the results. Effectively, all transformed benchmarks were equivalent to their original versions.

In a second phase of the experiment, we evaluated the performance. We strictly followed the methodology proposed in [32]. Each benchmark was repeatedly executed 1000 times in 100 different Java Virtual Machine (JVM) invocations (100.000 executions in total). To ensure real independence, the first iteration was always discarded (to avoid influence of dynamically loaded libraries persisting in physical memory, data persisting in the disk cache, etc.). From the 999 remaining iterations we retained 50 measurements per JVM invocation when steady-state performance was reached, i.e., once the coefficient of variation (CoV)[8] of the 50 iterations falls below a preset threshold of 0,01. Because 0,01 was difficult to reach for some benchmarks, we took the minimum CoV of 50 iterations in a row somewhere in between the window of 999 iterations. Then, for each JVM invocation, we computed the sum of the 50 benchmark iterations under steady-state performance. This produced one statistical value. With 100 JVM invocations, we obtained 100 statistical values. Finally, we computed the 0,99 confidence interval across the computed values from the different 100 JVM invocations.

This process was repeated for the 30 benchmarks (14 iterative versions + 14 recursive versions + 2 versions of *loops2recursion*). Iterative and recursive versions were executed interlaced, so that the impact of any possible variability in the overall system performance was minimized. This produced the set of measures that is shown in Table 5.9. Here, we use the notation $[_a\ b\ _c]$ that represents a symmetric 0,99 confidence interval between a and c with centre in b. Each column in the tables has the following meaning: `Benchmark` is the name of the benchmark. `Transf. Time` is the total amount of milliseconds needed by *loops2recursion* to produce the recursive version from the iterative version. Here, `Open` is the time needed to open all files of the target project, `Process` is the time needed to transform all loops in these files to recursion, and `Save` is the time needed to save all transformed files. `Loops` is the number of different loops in the source code. `Executed Loops` is the number of executed loops

[8]CoV is defined as the standard deviation s divided by the mean $\overline{x}$.

Benchmark	Transf. Time (ms)			Loops	Executed Loops	Executed Iterations
	Open	Process	Save			
loops2recursion	954	206	213	117	52580	153729
JGFArithBench	348	26	13	24	144	12590
JGFAssignBench	337	22	21	20	120	10389
JGFCastBench	161	16	6	8	48	4200
JGFCreateBench	337	27	17	34	520	44860
JGFExceptionBench	129	20	6	6	36	3148
JGFLoopBench	135	16	3	6	36	3159
JGFMathBench	419	45	33	60	360	31498
JGFMethodBench	212	20	9	16	96	8320
JGFCryptBench	146	20	8	10	277	4447
JGFFFTBench	158	17	8	14	4122	24570
JGFHeapSortBench	135	14	7	5	1502	12939
JGFSeriesBench	141	18	24	4	2005	1996013
JGFSORBench	155	21	4	7	10101	990102
JGFSparseMatmultBench	115	18	4	5	204	203200
Average	258,8	33,73	25,07	22,4	4810,07	233544,27

(a) Transformation performance

Benchmark	Iterative XT (ms)	Recursive XT (ms)	Rec/Iter XT (%)
loops2recursion	[7083,08 7195,94 7308,81]	[7023,41 7156,43 7289,46]	99,45 %
JGFArithBench	[144,72 144,96 145,20]	[167,49 167,59 167,69]	115,61 %
JGFAssignBench	[113,55 113,98 114,41]	[119,67 119,83 119,99]	105,14 %
JGFCastBench	[47,19 48,57 49,95]	[52,10 53,89 55,68]	110,94 %
JGFCreateBench	[13057,33 13155,91 13254,49]	[12422,11 12581,79 12741,48]	95,64 %
JGFExceptionBench	[819,71 824,62 829,52]	[9288,23 9292,87 9297,51]	1126,93 %
JGFLoopBench	[39,07 39,96 40,84]	[42,79 43,70 44,61]	109,34 %
JGFMathBench	[1616,55 1617,75 1618,95]	[985,08 986,07 987,06]	60,95 %
JGFMethodBench	[94,88 95,23 95,58]	[99,26 99,38 99,49]	104,36 %
JGFCryptBench	[281,63 285,11 288,59]	[287,46 291,65 295,83]	102,29 %
JGFFFTBench	[543,14 560,15 577,15]	[557,91 575,31 592,72]	102,71 %
JGFHeapSortBench	[405,34 414,09 422,84]	[406,91 414,92 422,93]	100,20 %
JGFSeriesBench	[44288,14 44363,17 44438,21]	[44033,69 44197,43 44361,16]	99,63 %
JGFSORBench	[7384,92 7426,96 7469,01]	[7466,40 7472,71 7479,01]	100,62 %
JGFSparseMatmultBench	[2149,15 2204,96 2260,77]	[2208,14 2273,53 2338,92]	103,11 %
Average	5232,76	5715,14	169,13 %

(b) Transformed code performance: Execution time results

Benchmark	Iterative LOT (ms)	Recursive LOT (ms)	Rec/Iter LOT (%)
loops2recursion	[1728,64 1754,99 1781,35]	[1804,54 1840,76 1876,98]	104,89 %
JGFArithBench	[42,44 42,52 42,60]	[49,31 49,37 49,42]	116,10 %
JGFAssignBench	[35,49 35,60 35,71]	[38,87 38,94 39,02]	109,38 %
JGFCastBench	[14,45 14,67 14,90]	[17,11 17,55 17,98]	119,57 %
JGFCreateBench	[96,97 97,42 97,87]	[121,28 123,59 125,90]	126,87 %
JGFExceptionBench	[10,60 11,29 11,97]	[12,86 12,90 12,95]	114,32 %
JGFLoopBench	[11,28 11,44 11,59]	[13,00 13,17 13,35]	115,19 %
JGFMathBench	[171,27 171,73 172,20]	[124,68 125,04 125,41]	72,81 %
JGFMethodBench	[28,18 28,29 28,40]	[30,62 30,67 30,72]	108,42 %
JGFCryptBench	[14,66 15,46 16,26]	[16,94 17,94 18,94]	116,06 %
JGFFFTBench	[73,44 79,44 85,44]	[88,51 94,96 101,41]	119,54 %
JGFHeapSortBench	[45,78 48,08 50,38]	[48,33 50,41 52,49]	104,84 %
JGFSeriesBench	[3952,32 3965,49 3978,66]	[3985,82 4000,19 4014,57]	100,88 %
JGFSORBench	[1841,56 1852,23 1862,91]	[1943,41 1944,21 1945,01]	104,97 %
JGFSparseMatmultBench	[606,37 634,00 661,62]	[660,63 695,42 730,22]	109,69 %
Average	584,18	603,68	109,57 %

(c) Transformed code performance: LOT results

Table 5.9: Results comparing iterative and recursive benchmark versions

(possibly with repetitions, especially in the case of nested loops). `Executed Iterations` is the number of loop iterations performed during the execution. The execution time (`XT`) is measured in milliseconds and represents the execution time of the whole benchmark. `LOT` stands for loop overhead time, it is measured in milliseconds, and it represents the time used to execute all loops without considering the time used by the body of the loops. That is, it just measures the time spent in a loop to control the own loop. This was calculated by inserting the `System.nanotime()` command, which returns the current time in nanoseconds, immediately before and after the loop, immediately after entering the iteration, and immediately before leaving the iteration.

Example 5.3.3 *Consider the following code that has been instrumented to measure its LOT:*

```
t1 = System.nanotime();
while (cond){
    t2 = System.nanotime();
    body;
    t3 = System.nanotime();
}
t4 = System.nanotime();
```

If we assume that the loop performs 0 *iterations, then LOT is* `t4 - t1`*.*
If we assume that the loop performs $n > 0$ *iterations, then LOT is calculated as follows:*

```
EntryTime = t2 - t1;  PrepareNextIterationTime = t2 - t3;  ExitTime = t4 - t3;
LOT = EntryTime + ((n - 1) * PrepareNextIterationTime) + ExitTime
```

Note that LOT is the appropriate measure to know what the real speedup reached with the transformation is. XT is not a good indicator because it depends on the body of the loops, but LOT is independent of the size and content of the loops.

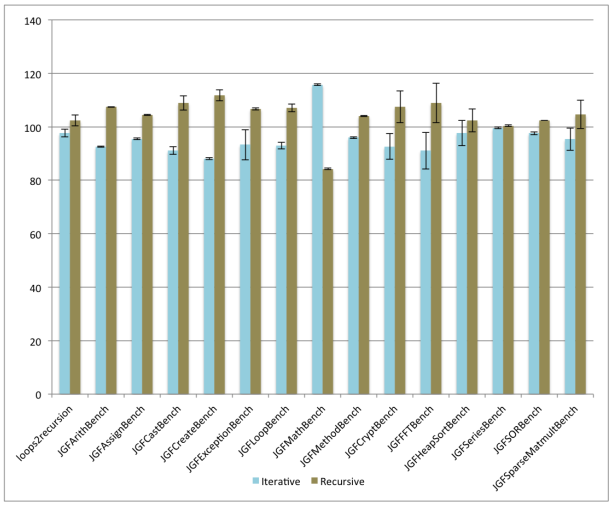
For the recursive versions, LOT is calculated in a similar way. For instance, in the `while-loop` row of Table 5.3:

- `t1 = System.nanotime();` is placed immediately after `Code 1`,
- `t2 = System.nanotime();` is placed immediately before `Code 2`,
- `t3 = System.nanotime();` is placed immediately after `Code 2`,
- `t4 = System.nanotime();` is placed immediately before `Code 3`.

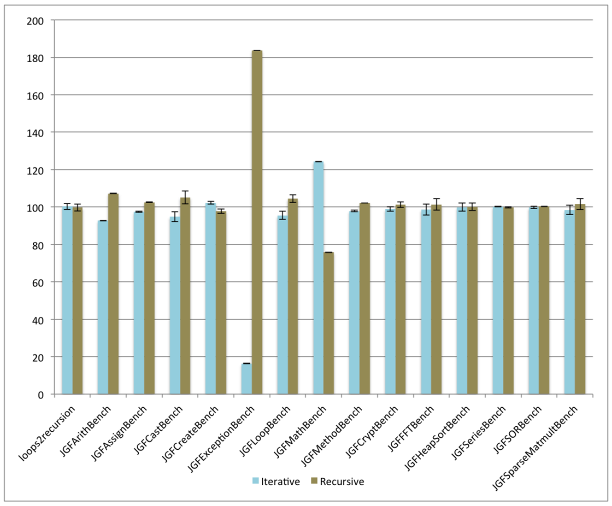
`Rec/Iter` measures the speedup/slowdown of the transformed code with respect to the original code. It is computed as 100×(Recursive time / Iterative time).

The results of LOT have been normalized in a chart shown in Figure 5.16(a). Thanks to the confidence intervals computed for the iterative and recursive versions we can extract some statistical conclusions. In 3 out of 15 experiments, the confidence intervals overlap, thus no statistical conclusion can be extracted. In 12 out of 15 experiments confidence intervals are disjoint. In 11 experiments, we can ensure with a confidence level of 99 % that the recursive version is less efficient than the iterative version of the loops. Finally, in 1 experiment we can ensure with a confidence level of 99 % that the recursive version is more efficient than the iterative version. Looking at Table 5.9, we see that the differences have been quantified and they are small (around 10 %).

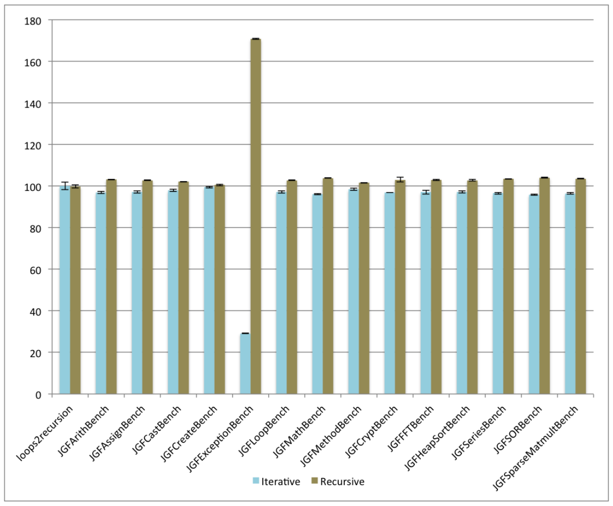
Initially, this was the end of our experiments. But then, we also looked at the results obtained for the XT. These results have been normalized in the chart shown in Figure 5.16(b). This figure reveals something unexpected: Sometimes the recursive version of the benchmarks is statistically (with 99 % confidence level) more efficient than the iterative version. One could think that this does not make sense, because all the code except LOT is exactly the same in the iterative version and in the recursive version. But it is the same *before compilation*. After compilation it is different due to the optimizations done by Java, which are different in both versions due to the presence of the transformed code. In order to prove this idea, we repeated all the experiments in interpreted-only mode using the '`-Xint`' JVM option. This flag disables the just-in-time (JIT) compilation in the JVM so that all bytecodes are executed by the interpreter. Disabling JIT compilation avoids many Java optimizations performed



(a) LOT results



(b) XT results



(c) XT results with -Xint flag enabled

Figure 5.16: Run-time charts that compare loops with recursion

to the code that significantly speed up recursion. The results obtained for XT with JIT compilation disabled are shown in Figure 5.16(c). The difference is clear. Without JIT compilation, iteration is more efficient. With JIT compilation iteration and recursion are similar, and sometimes recursion is even better (statistically). This confirms that in some cases, the library can increase code efficiency. However, in Java, our transformation has limited improvement due to the fact that the JVM does not support tail recursion optimization. This raises the point whether targeting another language the transformed code would produce better results. This part seems to be certainly interesting and it could open a line of research to decide in what cases the compiler should transform iterative loops into recursive methods to produce more efficient code. We left this research for future work.

5.3.7 Related work

The theoretical relation between iteration and recursion has been studied for decades in different paradigms, with regard to types, with regard to procedures, from a mathematical point of view, from an algorithmic point of view, etc. There is a big amount of work that studies this theoretical relation (see, e.g., [34, 29]).

On the practical side, there exist many different approaches to transform recursive functions into loops (see, e.g., [42, 62, 71]). This has been in fact a hot area of research related to compiler optimization and tuning. Contrarily, there are very few approaches devoted to transform loops into equivalent recursive functions. However, recursion provides important advantages in debugging [56], theorem proving [70], verification [74] and termination analysis [27].

For instance, this transformation can be very useful in AD. In this debugging technique, the final output of the debugger is always a buggy method. This is often very imprecise and forces the user to inspect the whole method to find the bug. It is possible that the method implements several loops with their own functionality. If we are able to transform all loops into recursive methods, this would mean that algorithmic debuggers would be able to discard or detect bugs inside these loops, and even report a particular loop as buggy. For this reason, the debugger DDJ implements our transformation from loops to recursion since version 2.6.

In some articles, some transformations are proposed or mentioned for particular types of loops. For instance, in [27] a transformation for *while*-loops is used to detect the termination conditions of the transformed recursive functions. Unfortunately, the other types of loops are ignored, but they provide a treatment for the *break* statement (however, they do not allow for the use of labels, which is what introduces the complexity in the transformation).

In [104], authors argue that recursive functions are more efficient than loops in some architectures. In particular, they want to measure the performance benefits of recursive algorithms on multi-level memory hierarchies and on shared memory systems. For that, they define a transformation for *do*-loops (very similar to the Java's *for*-loop).[9] The use of *break* and *continue* is not accepted by the transformation.

Myreen and Gordon describe a theorem prover in [74]. In their examples, they include an ad-hoc transformation of a *while*-loop into an equivalent recursive function. This is done because the recursive function facilitates the verification condition generation and avoids the inclusion of assertions in the source code.

We have not been able to find a description of how to transform some specific loops such as the *foreach*-loop. Moreover, the program transformations that appear in the literature are just mentioned or based on ad-hoc examples; and we are not aware of any transformation that accepts *break/continue* statements with labels. The use of exceptions has been also ignored in previous implementations. To the best of our knowledge no systematic transformation has been described in the literature yet, thus, researchers and programmers are condemned to reinvent the wheel once and again.

The lack of a systematic program transformation is a source of inefficient implementations. This happens for instance when transforming loops into functions that are not tail-recursive [104] or in implementations that only work for a reduced subset of the languages and produce buggy transformed code in presence of not treated commands such as *throws*, *try-catch*, *break* or *continue* [30].

In this work, we also describe our implementation that is the first to accept the whole Java language; and it allows us (in the particular case) to automatically transform a given loop into a set of equivalent recursive methods, or (in the general case) to input a program and output an equivalent program where all loops have been automatically transformed into recursion. This library has been already incorporated in the Declarative Debugger for Java (DDJ) [46].

Even though we present a transformation for Java, it could be easily adapted to many other languages. The only important restriction is that `goto` statements are not treated (we discussed their use in Section 5.3.5). Moreover, our transformation uses some features of Java:

- it uses blocks to define scopes. Those languages where blocks cannot be defined, or where the scope is not limited by the use of blocks, should use fresh variable names in the generated code to avoid variable clashes between the variables of the transformation and the variables of the rest of the program.

[9] The syntax of their *do*-loops is "`do k = 1, N`" meaning that `N` iterations must be done with `k` taking values from `1` to `N`.

- it assumes that, by default, arguments are passed by value. In those languages where the default argument passing mechanism is different, the transformation can be simplified. For instance, with call by value, exceptions need a special treatment described in Section 5.3.4. With call by reference, this special treatment can be omitted.

5.3.8 Proofs of technical results

In this section we provide a formal semantics-based specification of our transformation in order to prove the correctness of the basic transformation (we skip exceptions and labels in this section). For this, we provide a BNF syntax specification and an operational semantics of Java. We consider the subset of Java that is needed to implement the transformation (*if-then-else*, *while*, method calls, *return*, etc.), and we ignore the rest of syntax elements for the sake of simplicity (they do not have any influence because any command inside the body of the loop remains unchanged in the transformed code). Moreover, in this section, we centre the discussion on *while*-loops and prove properties for this type of loop. The proof for the other types of loops is omitted, but it would be in all cases analogous or slightly incremental.

We start with a BNF syntax of Java:

P	$::=$	$M_1, \ldots, M_n, S_p$	(program)	*Domains*
				$x, y, z \ldots \in \mathbb{V}$ (variables)
				$a, b, c \ldots \in \mathbb{C}$ (constants)
M	$::=$	$m(x_1, \ldots, x_n)$ { S_p; S_r }	(method definition)	where $x_1, \ldots, x_n \in \mathbb{V}$ and
				m is the name of the method
S_p	$::=$	$x := E$	(assignment)	
	$\mid$	$x := m(E_0, \ldots, E_n)$	(method invocation)	
	$\mid$	if E_b then S_p	(if-then)	
	$\mid$	if E_b then S_p else S_p	(if-then-else)	
	$\mid$	while E_b do S_p	(while)	
	$\mid$	S_p; S'_p	(sequence)	
S_r	$::=$	return E	(return)	
E	$::=$	$E_a \mid E_b$	(expresion)	
E_a	$::=$	$E_a + E_a \mid E_a - E_a \mid V$	(arithmetic expresion)	
E_b	$::=$	$E_b \mathrel{!=} E_b \mid E_b == E_b \mid V$	(boolean expresion)	
V	$::=$	$x \mid a$	(variables or constants)	

A program is a set of method definitions and at least one initial statement (usually a method invocation). Each method definition is composed of a set of statements followed by a *return* statement. For simplicity, the arguments of a method invocation can only be expressions (not statements). This is not a restriction, because any statement can be assigned to a variable and then be passed as argument of the method invocation. However, this simplification allows us to ease the semantics of method invocations and, thus, it increases readability.

In the following we consider two versions of the same program shown in Algorithms 17 and 18. We assume that in Algorithm 17 there exists a x variable already defined before the loop and, for the sake of simplicity, it is the only variable modified inside S. Therefore, Algorithm 18 is the recursive version of the *while*-loop in Algorithm 17 according to our transformation, and hence, $p_0, \ldots, p_n$ represent all variables defined before the loop and used in S (the loop statements) and *cond* (the loop condition). In the case that more than one variable are modified, then the output would be an array with all modified variables. We avoid this case because it is not necessary for the proof.

Algorithm 17 While version

```
while $cond$ do
    $S$;
```

Algorithm 18 Recursive version

```
$m(p_0, \dots, p_n)$ {
    $S$;
    if $cond$ then
        $x := m(a_0, \dots, a_n)$;
    return $x$;
}
if $cond$ then
    $x := m(a_0, \dots, a_n)$;
```

In order to provide an operational semantics for this Java subset, which allows for recursion, we need a stack to push and pop different frames that represent individual method activations. Frames, $f_0, f_1, \ldots \in \mathbb{F}$, are sequences of variable-value pairs. States, $s_0, s_1, \ldots \in \mathbb{S}$, are sequences of frames ($\mathbb{S} : \ \mathbb{F}$ x ... x $\mathbb{F}$). We make the program explicitly accessible to the semantics through the use of an environment, $e \in \mathbb{E}$, represented with a sequence of functions from method names $\mathbb{M}$ to pairs of parameters $\mathbb{P}$ and statements $\mathbb{I}$ ($\mathbb{E} : \ (\mathbb{M} \rightarrow (\mathbb{P}$ x $\mathbb{I}))$ x ... x $(\mathbb{M} \rightarrow (\mathbb{P}$ x $\mathbb{I})))$. Our semantics is based on the Java semantics described in [77] with some minor modifications. It uses a set of functions to update the state, the environment, etc.

The Upd_v function is used to update a variable (var) in the current frame of the state (s) with a value ($value$). The current frame in the state is always the last introduced frame (i.e., the last element in the sequence of frames that represent the state). We use the standard notation $f[var \rightarrow value]$ to denote that variable var in frame f is updated to the $value$ value.

$$Upd_v(s,\ var \rightarrow value) = \begin{cases} error & if\ s = [] \\ [f_0, \ldots, f_n[var \rightarrow value]] & if\ s = [f_0, \ldots, f_n] \end{cases}$$

The Upd_r function records the returned value ($value$) of the current frame of the state (s) inside a fresh variable $\Re$ of this frame, so that other frames can consult the value returned by the current frame.

$$Upd_r(s,\ value) = \begin{cases} error & if\ s = [] \\ [f_0, \ldots, f_n[\Re \rightarrow value]] & if\ s = [f_0, \ldots, f_n] \end{cases}$$

The Upd_{vr} function is used to update a variable (var) in the penultimate frame of the state (s) taking the value returned by the last frame in the state (which must be previously stored in $\Re$). This happens when a method calls another method and the latter finishes returning a value. In this situation, the last frame in the state should be removed and the returned value should be updated in the penultimate frame. We use the notation $f_n(\Re)$ to consult the value of the $\Re$ variable in the f_n frame.

$$Upd_{vr}(s,\ var) = \begin{cases} error & if\ s = []\ or\ s = [f] \\ [f_0, \ldots, f_{n-1}[var \rightarrow f_n(\Re)], f_n] & if\ s = [f_0, \ldots, f_{n-1}, f_n] \end{cases}$$

The Upd_e function is used to update the environment (env) with a new method definition ($m \rightarrow (P, I)$). The environment is used in method invocations to know the method that should be executed.

$$Upd_e(env,\ m \rightarrow (P, I)) = env[m \rightarrow (P, I)]$$

The *AddFrame* function adds a new frame to the state (s). This frame is a sequence of mappings from parameters ($p_0, \ldots, p_n$) to the evaluation of arguments ($a_0, \ldots, a_n$). To evaluate an expression we use the *Eval* function: a variable is consulted in the state, a constant is just returned, and a mathematical or boolean expression is evaluated with the standard semantics. We use this notation because the evaluation of expressions does not have influence in our proofs, but it significantly reduces the size of derivations, thus, improving the clarity of the presentation.

$$AddFrame(s,[p_0,\ldots,p_n],[a_0,\ldots,a_n]) = \begin{cases} [[p_0 \rightarrow Eval(a_0),\ldots,p_n \rightarrow Eval(a_n)]] & if\ s=[] \\ [f_0,\ldots,f_m,[p_0 \rightarrow Eval(a_0),\ldots,p_n \rightarrow Eval(a_n)]] & if\ s=[f_0,\ldots,f_m] \end{cases}$$

Analogously, the *RemFrame* function removes the last frame inserted into the state (s).

$$RemFrame(s) = \begin{cases} error & if\ s=[] \\ [] & if\ s=[f] \\ [f_0,\ldots,f_n] & if\ s=[f_0,\ldots,f_n,f_{n+1}] \end{cases}$$

We are now in a position to introduce our Java operational semantics. Essentially, the semantics is a big-step semantics composed of a set of rules of the form: $\frac{p_1 \ldots p_n}{env \vdash <st,\ s> \Downarrow s'}$ that should be read as "The execution of the *st* statement in the *s* state under the *env* environment can be reduced to the s' state provided that the $p_1 \ldots p_n$ premises hold". The rules of the semantics are shown below.

New method

$$\frac{env' = Upd_e(env,\ m_0 \rightarrow (P,\ I)) \wedge env' \vdash <m_1\ i,\ s> \Downarrow s'}{env \vdash <m_0(P)\{I\}\ m_1\ i,\ s> \Downarrow s'}$$

Empty statement

$$\frac{}{env \vdash <\surd,\ s> \Downarrow s}$$

Asignment

$$\frac{s' = Upd_v(s,\ x \rightarrow Eval(op,\ s))}{env \vdash <x{:=}op,\ s> \Downarrow s'}$$

Method invocation

$$\frac{(P,\ I) = env(m) \wedge s' = AddFrame(s,P,A) \wedge env \vdash <I,\ s'> \Downarrow s'' \wedge s''' = Upd_{vr}(s'',x) \wedge s'''' = RemFrame(s''')}{env \vdash <x{:=}m(A),\ s> \Downarrow s''''}$$

If

$$\frac{env \vdash <if\ cond\ then\ i_0\ else\ \surd,\ s> \Downarrow s'}{env \vdash <if\ cond\ then\ i_0,\ s> \Downarrow s'}$$

$$\frac{<cond,\ s> \Rightarrow true \wedge env \vdash <i_0,\ s> \Downarrow s'}{env \vdash <if\ cond\ then\ i_0\ else\ i_1,\ s> \Downarrow s'}$$

$$\frac{<cond,\ s> \Rightarrow false \wedge env \vdash <i_1,\ s> \Downarrow s'}{env \vdash <if\ cond\ then\ i_0\ else\ i_1,\ s> \Downarrow s'}$$

While

$$\frac{<cond,\ s> \Rightarrow false}{env \vdash <while\ cond\ do\ i,\ s> \Downarrow s}$$

$$\frac{<cond,\ s> \Rightarrow true \wedge env \vdash <i,\ s> \Downarrow s' \wedge env \vdash <while\ cond\ do\ i,\ s'> \Downarrow s''}{env \vdash <while\ cond\ do\ i,\ s> \Downarrow s''}$$

Sequence

$$\frac{env \vdash <i_0,\ s> \Downarrow s' \wedge env \vdash <i_1,\ s'> \Downarrow s''}{env \vdash <i_0;\ i_1,\ s> \Downarrow s''}$$

Return

$$\frac{s' = Upd_r(s,\ Eval(op,\ s))}{env \vdash <return\ op,\ s> \Downarrow s'}$$

We can now prove our main result.

Theorem 5.3.4 (Correctness) *Algorithm 18 is semantically equivalent to Algorithm 17.*

Proof. We prove this claim by showing that the final state of Algorithm 17 is always the same as the final state of Algorithm 18. The semantics of a program P is:

$$S(P) = s \quad \text{iff} \quad [\,] \vdash < P,\ [\,] > \Downarrow s$$

Therefore, we say that two programs P_1 and P_2 are equivalent if they have the same semantics:

$$S(P_1) = S(P_2) \quad \text{iff} \quad [\,] \vdash < P_1,\ [\,] > \Downarrow s_1 \quad \wedge \quad [\,] \vdash < P_2,\ [\,] > \Downarrow s_2 \quad \wedge \quad s_1 = s_2$$

For the sake of generality, in the following we consider that the loops can appear inside any other code. Therefore, the environment and the state are not necessarily empty. Thus, we will assume an initial environment env_0 and an initial state s: $env_0 \vdash < P,\ s >$. We prove this semantics equivalence analyzing two possible cases depending on whether the loop is executed or not.

1) <u>Zero iterations</u>

This situation can only happen when the *cond* condition is not satisfied the first time it is evaluated. Hence, we have the following semantics derivation for each program:

<u>Iterative version</u>

$$\frac{< cond,\ s > \ \Rightarrow\ false}{env_0 \ \vdash\ < while\ cond\ do\ S,\ s > \ \Downarrow\ s}$$

<u>Recursive version</u>

$$\frac{env = Upd_e(env_0,\ m\ \rightarrow\ (P,\ S; I)) \qquad \dfrac{\dfrac{< cond,\ s > \ \Rightarrow\ false \quad \dfrac{}{env\ \vdash\ < \surd,\ s > \ \Downarrow\ s}}{env\ \vdash\ < if\ cond\ then\ t\ else\ \surd,\ s > \ \Downarrow\ s}}{env\ \vdash\ < if\ cond\ then\ t,\ s > \ \Downarrow\ s}}{env_0\ \vdash\ < m(P)\ \{\ S; I\ \}\ if\ cond\ then\ t,\ s > \ \Downarrow\ s}$$

Clearly, the state is never modified, neither in the iterative version nor in the recursive version. Therefore, both versions are semantically equivalent.

2) <u>One or more iterations</u>

This means that the *cond* condition is satisfied at least once. Let us consider that *cond* is satisfied n times, producing n iterations. We proof that the final state of the program in Algorithm 18 is equal to the final state of the program in Algorithm 17 by induction over the number of performed iterations.

(Base Case) In the base case, only one iteration is executed. Hence, we have the following derivations:

<u>Iterative version</u>

$$\frac{< cond,\ s > \ \Rightarrow\ true \quad env_0\ \vdash\ < S,\ s > \ \Downarrow\ s^1 \quad \dfrac{< cond,\ s > \ \Rightarrow\ false}{env_0\ \vdash\ < while\ cond\ do\ S,\ s^1 > \ \Downarrow\ s^1}}{env_0\ \vdash\ < while\ cond\ do\ S,\ s > \ \Downarrow\ s^1}$$

<u>Recursive version</u>

$$\dfrac{\dfrac{env \vdash\ < S,\ s^1 >\ \Downarrow\ s^2 \qquad \dfrac{\dfrac{\dfrac{< cond,\ s^2 >\ \Rightarrow\ false \qquad \overline{env \vdash\ < \surd,\ s^2 >\ \Downarrow\ s^2}}{env \vdash\ < if\ cond\ then\ t\ else\ \surd,\ s^2 >\ \Downarrow\ s^2}}{env \vdash\ < if\ cond\ then\ t,\ s^2 >\ \Downarrow\ s^2} \qquad \dfrac{s^3\ =\ Upd_r(s^2,\ Eval(x, s^2))}{env \vdash\ < return\ x,\ s^2 >\ \Downarrow\ s^3}}{env \vdash\ < if\ cond\ then\ t;\ return\ x,\ s^2 >\ \Downarrow\ s^3}}{env \vdash\ < S; I,\ s^1 >\ \Downarrow\ s^3}}{\triangle}$$

$$\dfrac{\dfrac{< cond,\ s >\ \Rightarrow\ true \qquad \dfrac{(P, I) = env(m) \quad s^1\ =\ AddFrame(s, P, [a_0, \ldots, a_n]) \quad \triangle \quad s^4\ =\ Upd_{vr}(s^3, x) \quad s^5\ =\ RemFrame(s^4)}{env \vdash\ < x := m(a_0, \ldots, a_n),\ s >\ \Downarrow\ s^5}}{env \vdash\ < if\ cond\ then\ t\ else\ \surd,\ s >\ \Downarrow\ s^5}}{env \vdash\ < if\ cond\ then\ t,\ s >\ \Downarrow\ s^5}$$

We can assume that the x variable has an initial value z^0, which must be the same in both versions of the algorithm. Then, states are modified during the iteration as follows:

<u>Iterative version</u>	<u>Recursive version</u>
$s = [f_0] \Rightarrow f_0 = \{x \rightarrow z^0\}$	$s = [f_0] \Rightarrow f_0 = \{x \rightarrow z^0\}$
$s^1 = [f_0] \Rightarrow f_0 = \{x \rightarrow z^1\}$	$s^1 = [f_0, f_1] \Rightarrow f_0 = \{x \rightarrow z^0\} \wedge f_1 = \{x \rightarrow z^0\}$
	$s^2 = [f_0, f_1] \Rightarrow f_0 = \{x \rightarrow z^0\} \wedge f_1 = \{x \rightarrow z^1\}$
	$s^3 = [f_0, f_1] \Rightarrow f_0 = \{x \rightarrow z^0\} \wedge f_1 = \{x \rightarrow z^1, \Re \rightarrow z^1\}$
	$s^4 = [f_0, f_1] \Rightarrow f_0 = \{x \rightarrow z^1\} \wedge f_1 = \{x \rightarrow z^1, \Re \rightarrow z^1\}$
	$s^5 = [f_0] \Rightarrow f_0 = \{x \rightarrow z^1\}$

Clearly, with the same initial states, both algorithms produce the same final state.

(Induction Hypothesis) We assume as the induction hypothesis that executing i iterations in both versions with an initial value z^0 for x then, if the iterative version obtains a final value z^n for x then the recursive version correctly obtains and stores the same final value z^n for variable x in the top frame.

<u>Iterative version</u>	<u>Recursive version</u>
$s = [f_0] \Rightarrow f_0 = \{x \rightarrow z^0\}$	$s = [f_0] \Rightarrow f_0 = \{x \rightarrow z^0\}$
...	...
$s' = [f_0] \Rightarrow f_0 = \{x \rightarrow z^n\}$	$s' = [f_0] \Rightarrow f_0 = \{x \rightarrow z^n\}$

(Inductive Case) We now prove that executing $i + 1$ iterations in both versions with an initial value z^0 for x then, if the iterative version obtains a final value z^n for x then the recursive version correctly obtains and stores the same final value z^n for variable x in the top frame.

The derivation obtained for each version is the following:

<u>Iterative version</u>

$$\dfrac{< cond,\ s >\ \Rightarrow\ true \quad env_0\ \vdash\ < S,\ s >\ \Downarrow\ s^1 \quad \dfrac{Induction\ hypotesis}{env_0\ \vdash\ < while\ cond\ do\ S,\ s^1 >\ \Downarrow\ s^2}}{env_0\ \vdash\ < while\ cond\ do\ S,\ s >\ \Downarrow\ s^2}$$

Recursive version

$$\cfrac{env \vdash \ < S,\ s^1 > \ \Downarrow \ s^2 \qquad \cfrac{\cfrac{Induction\ hypotesis}{env \vdash \ < if\ cond\ then\ t,\ s^2 > \ \Downarrow \ s^3} \qquad \cfrac{s^4 \ = \ Upd_r(s^3,\ Eval(x, s^3))}{env \vdash \ < return\ x,\ s^3 > \ \Downarrow \ s^4}}{env \vdash \ < if\ cond\ then\ t;\ return\ x,\ s^2 > \ \Downarrow \ s^4}}{\cfrac{env \vdash \ < S; I,\ s^1 > \ \Downarrow \ s^4}{\triangle}}$$

$$\cfrac{< cond,\ s > \ \Rightarrow \ true \qquad \cfrac{(P, I) = env(m) \quad s^1 \ = \ AddFrame(s, P, [a_0, \ldots, a_n]) \quad \triangle \quad s^5 \ = \ Upd_{vr}(s^4, x) \quad s^6 \ = \ RemFrame(s^5)}{env \vdash \ < x := m(a_0, \ldots, a_n),\ s > \ \Downarrow \ s^6}}{\cfrac{env \vdash \ < if\ cond\ then\ t\ else\ \surd,\ s > \ \Downarrow \ s^6}{env \vdash \ < if\ cond\ then\ t,\ s > \ \Downarrow \ s^6}}$$

Because both algorithms have the same initial value z^0 for x then the states during the iteration are modified as follows (the * state is obtained by the induction hypothesis):

Iterative version	Recursive version
$s = [f_0] \Rightarrow f_0 = \{x \to z^0\}$	$s = [f_0] \Rightarrow f_0 = \{x \to z^0\}$
$s^1 = [f_0] \Rightarrow f_0 = \{x \to z^1\}$	$s^1 = [f_0, f_1] \Rightarrow f_0 = \{x \to z^0\} \wedge f_1 = \{x \to z^0\}$
$s^2 = [f_0] \Rightarrow f_0 = \{x \to z^n\}*$	$s^2 = [f_0, f_1] \Rightarrow f_0 = \{x \to z^0\} \wedge f_1 = \{x \to z^1\}$
	$s^3 = [f_0, f_1] \Rightarrow f_0 = \{x \to z^0\} \wedge f_1 = \{x \to z^n\}*$
	$s^4 = [f_0, f_1] \Rightarrow f_0 = \{x \to z^0\} \wedge f_1 = \{x \to z^n, \Re \to z^n\}$
	$s^5 = [f_0, f_1] \Rightarrow f_0 = \{x \to z^n\} \wedge f_1 = \{x \to z^n, \Re \to z^n\}$
	$s^6 = [f_0] \Rightarrow f_0 = \{x \to z^n\}$

Hence, Algorithm 18 and Algorithm 17 obtain the same final state, and thus, they are semantically equivalent. $\square$

5.4 Loop Expansion

5.4.1 Introduction

In this section we present a technique called *Loop Expansion* (LE) [56] to improve the performance of Algorithmic Debugging (AD) by reducing the debugging time. Let us explain the technique with an example. Consider the Java program shown in Figure 5.17.

```
public class Matrix {
   private int numRows;
   private int numColumns;
   private int[][] matrix;

(1)public Matrix(int numRows, int numColumns) {
      this.numRows = numRows;
      this.numColumns = numColumns;
      this.matrix = new int[numRows][numColumns];
      for (int i = 0; i < numRows; i++)
         for (int j = 0; j < numColumns; j++)
            this.matrix[i][j] = 1;
   }

(2)public int position(int numRow, int numColumn) {
      return matrix[numRow][numColumn];
   }
}

public class SumMatrix {
(0)public static void main(String[] args) {
      int result = 0;
      int numRows = 3;
      int numColumns = 3;
      Matrix m = new Matrix(numRows, numColumns);
      for (int i = 0; i < numRows; i++)
         for (int j = 0; j < numColumns; j++)
            result += m.position(i, j);
      System.out.println(result);
   }
}
```

Figure 5.17: Matrix Java program

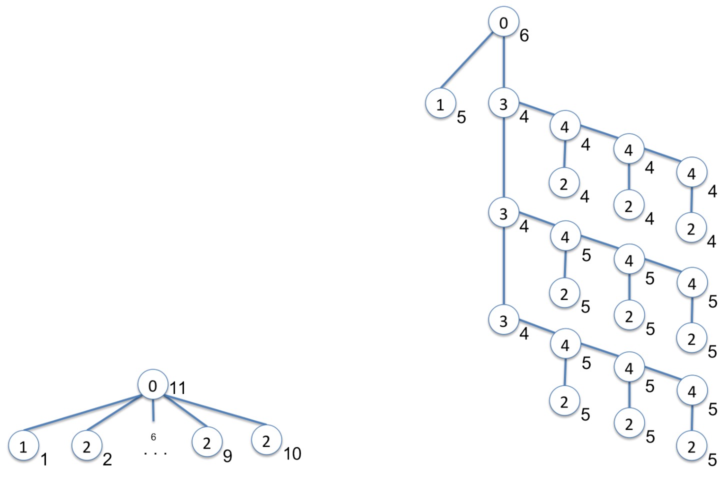


Figure 5.18: ETs of the examples in Figure 5.17 (left) and 5.19 (right)

This program initializes the elements of a matrix to 1, and then traverses the matrix to add all of them up. The Execution Tree (ET) associated with this example is shown in Figure 5.18 (left). Because it is not relevant in loop expansion, in the rest of this section we omit the information of the nodes in the ETs, and instead we label them (inside) with the number of the method associated with them. Observe that `main` (0) calls once to the constructor `Matrix` (1), and then calls nine times to `position` (2) inside two nested loops. In the ET, every node has a number (outside) that indicates the number of questions needed to find the bug when it is the buggy node. To compute this number, we have considered the Divide and Query [92] strategy, that always selects the ET node that minimizes the difference between the numbers of descendants and non-descendants (i.e., better divides the ET in half).

AD can produce long series of questions making the debugging session too long. Moreover, it works at the level of methods, thus the granularity level of the located bug is a method. Loop Expansion is particularly interesting to solve these problems because it can help to reduce the complexity of the questions, and also the granularity level of the located bug. This technique is based on a transformation of the source code, and it produces an ET that can be debugged more efficiently using the standard algorithms. Therefore, LE is conservative with respect to previous implementations and it can be integrated in any debugger as a preprocessing stage. The cost of the transformation is low compared with the cost of generating the whole ET. In fact, it is efficient enough as to be always used in all algorithmic debuggers before the ET exploration phase.

The rest of the section has been organized as follows. Section 5.4.2 introduces some preliminary definitions that will be used in the rest of the section. In Section 5.4.3 we explain LE and its main applications. The correctness of LE is proved in Section 5.4.4. Finally, in Section 5.4.5 we discuss the related work.

5.4.2 Preliminaries

In this section, we use numbers as method identifiers that uniquely identify each method in the source code. This simplification is enough to keep our definitions and algorithms precise and simple. Given an ET, the identifier of the method associated with a node n is referenced with $l(n)$, *simple recursion* is represented with a branch of nodes with the same identifier. *Nested recursion* happens when a recursive branch is descendant of another recursive branch. *Multiple recursion* happens when a node labelled with an identifier n has two or more children labelled with n.

5.4.3 Loop Expansion optimization

Expanding loops into recursive calls is very useful for AD. We, can see an example of this in the right ET of Figure 5.18. Here, the nodes labelled with 3 inside are the nodes generated from the recursive calls, and thus each of them is representing an iteration of the loop. Observe that each recursive node of the recursive branch is the root of a particular subtree (on their right). This is very convenient because it allows the debugger to prune different iterations. In contrast, loops produce very wide trees where all iterations are represented as trees with a common root (see the left tree of Figure 5.18). In this structure, it is impossible to prune more than one iteration at a time, being the debugging of these trees very expensive.

To perform this optimization, in this section we present a technique for AD that transforms loops into equivalent recursive methods. Because current compilers produce more efficient code for iteration than for recursion, there exist many approaches to transform recursive methods into equivalent loops (e.g., [42, 62]). However, there exist few approaches to transform loops into equivalent recursive methods. An exception is the one presented in [104] to improve performance in multi-level memory hierarchies. Nevertheless, we are not aware of any algorithm of this type proposed for Java or for any other object-oriented language. Hence, we had to implement this algorithm as a Java library and made it public for the community:

`http://www.dsic.upv.es/~jsilva/loops2recursion/`

Our library to transform iterative loops into recursive methods is called *loops2recursion* (see Section 5.3). This library has been integrated into an algorithmic debugger [46] as part of LE. This algorithm has an asymptotic cost linear with respect to the number of loops in the program and it is the basis of LE. Basically, it transforms each loop into an equivalent recursive method. The transformation is slightly different for each type of loop (`while`, `do`, `for`, or `foreach`). In the case of `for`-loops, it can be explained with the code in Figure 5.20 where A, B, C and D represent blocks of code. If we observe the transformed ET, where the information shown for each node is the call to the function, we see that each iteration is represented with a different node of the recursive branch $r(1) \to r(2) \to \ldots \to r(10)$, thus it is possible to detect a bug in a single iteration. This means that, in the case that the f function had a bug, thanks to the transformation, the debugger could detect that a bug exists in the code in B + C or in A + D. Note that this is not possible in the original ET where the debugger would report that A + B + C + D has a bug. This is a very important result because it augments the granularity level of the reported bugs, detecting bugs inside loops and not only inside methods.

Example 5.4.1 *An AD session for the right ET in Figure 5.20 using D&Q follows (*`YES` *and* `NO` *answers are provided by the user):*

```
Starting Debugging Session...
r(5) YES   r(3) YES   r(2) NO   g(2, ...) YES
Bug located in: the second iteration of the r loop in method f(...).
```

Thanks to the nodes inserted by LE, the debugger can identify that the bug is located in the code in B + C and not in A + D.

Nested recursion augments the possibilities of pruning. For instance, the Matrix class in Figure 5.17 can be automatically transformed[10] to the code in Figure 5.19. The ET obtained from executing the transformed program is shown in Figure 5.18 (right). Observe that there is a recursion branch for each executed loop, and thus, we have recursive branches (those labelled with 4) inside a recursive branch (labelled with 3). Hence, the new nodes added by the transformation are used to represent each single iteration; and thanks to them, now it is possible to prune loops, iterations, or single calls inside an iteration.

5.4.4 Correctness

In this section we prove that after our transformation, all bugs that could be detected in the original ET can still be detected in the transformed one. An even more interesting result is that the transformed ET can contain more buggy nodes than the original one, and thus, we can detect bugs that before were undetectable. The correctness of LE is stated in the following.

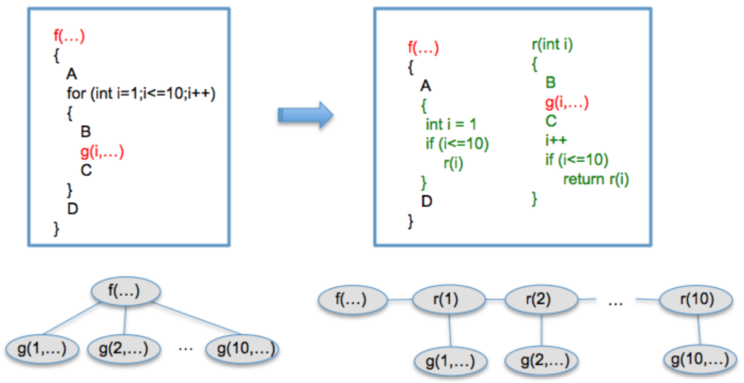
Theorem 5.4.2 (Completeness) *Let $\mathcal{P}$ be a program, let T be the ET associated with $\mathcal{P}$, let $\mathcal{P}'$ be the program obtained by applying Loop Expansion to $\mathcal{P}$, and let T' be the ET associated with $\mathcal{P}'$. For each buggy node in T, there is at least one buggy node in T'.*

Proof. Let us prove the theorem for an arbitrary buggy node n in T associated with a f function. Firstly, because LE only transforms iterative loops into recursive loops, all functions executed in P are

[10]For the sake of clarity, in the figure we replace the names of the generated recursive methods by `sumRows` and `sumColumns`. In the implementation, if a loop has a Java label in the original source code, the transformation uses this label to name the recursive method. If this label does not exist, then the name of the loop is the name of the method that contains this loop followed by "_loopN", where N is an autonumeric. While debugging, the user can see the source code of the loop, and they can change its name if they want to do it.

```
public class SumMatrix {
(0) public static void main(String[] args) {
      int result = 0;
      int numRows = 3;
      int numColumns = 3;
      Matrix m = new Matrix(numRows, numColumns);
      // For loop
      { // Init for loop
         int i = 0;
         // First iteration
         if (i < numRows) {
            Object[] res = SumMatrix.sumRows(m, i, numRows, numColumns, result);
            result = (Integer) res[0];
         }
      }
      System.out.println(result);
   }
(3) private static Object[] sumRows(Matrix m, int i, int numRows, int numColumns, int result) {
      // For loop
      { // Init for loop
         int j = 0;
         // First iteration
         if (j < numColumns) {
            Object[] res = SumMatrix.sumColumns(m, i, j, numColumns, result);
            result = (Integer) res[0];
         }
      }
      // Update for loop
      i++;
      // Next iteration
      if (i < numRows)
         return SumMatrix.sumRows(m, i, numRows, numColumns, result);
      return new Object[] { result };
   }
(4) private static Object[] sumColumns(Matrix m, int i, int j, int numColumns, int result) {
      result += m.position(i, j);
      // Update for loop
      j++;
      // Next iteration
      if (j < numColumns)
         return SumMatrix.sumColumns(m, i, j, numColumns, result);
      return new Object[] { result };
   }
}
```

Figure 5.19: Recursive version of the program in Figure 5.17



```
f(...)
{
  A
  for (int i=1;i<=10;i++)
  {
    B
    g(i,...)
    C
  }
  D
}
```

```
f(...)                r(int i)
{                     {
  A                     B
  {                     g(i,...)
   int i = 1            C
   if (i<=10)           i++
      r(i)              if (i<=10)
  }                        return r(i)
  D                   }
}
```

Figure 5.20: ET transformation from a loop into a recursive method

also executed in P'. This means that every node in T has a counterpart (equivalent) node in T' that represents the same (sub)computation. Therefore, we can call n' to the node that represents in T' the same execution than n in T. Because n is buggy, then n is wrong, and all the children of n (if any) are correct. Hence, n' is also wrong. Moreover, if f does not contain a loop, then LE has no effect on the code of f and thus n and n' will have exactly the same children, and thus, trivially, n' is also buggy in T'. If we assume the existence of a loop in f, then we will have a situation as the one shown in the ETs of Figure 5.20. We can consider for the proof that the ET on the left is the subtree of n and the ET on the right is the subtree of n'. Then, because n is buggy, all nodes labelled with g are correct (in both ETs) and, thus, either n' or one of the nodes labelled with r are buggy. $\square$

Theorem 5.4.3 (Soundness) *Let $\mathcal{P}$ be a program, let T be the ET associated with $\mathcal{P}$, let $\mathcal{P}'$ be the program obtained by applying Loop Expansion to $\mathcal{P}$, and let T' be the ET associated with $\mathcal{P}'$. If T contains a buggy node associated with code $f \subseteq \mathcal{P}$, then, T' contains a buggy node associated with code $g \subseteq \mathcal{P}$ and $g \subseteq f$.*

Proof. According to the proof of Theorem 5.4.2, every node in T has a counterpart (equivalent) node in T'. Hence, let n be the buggy node in T and let n' be the associated buggy node in T'. If f does not have a loop, then both n and n' point to the same function (f) and thus the theorem holds trivially. If f contains a loop that has been expanded, then, as stated in the proof of Theorem 5.4.2, either n' or one of its descendants (say n'') that represent the iterations of the loop are buggy. But we know that the code of n'' is the code of the loop that is included in the code of f. Therefore, assuming that g is the code of either n' or n'', then in all cases $g \subseteq f$. $\square$

From Theorem 5.4.2 and 5.4.3 we have a very interesting corollary that reveals that more bugs can be found in the transformed tree than in the original ET.

Corollary 5.4.4 *Let $\mathcal{P}$ be a program, let T be the ET associated with $\mathcal{P}$, let $\mathcal{P}'$ be the program obtained by applying Loop Expansion to $\mathcal{P}$, and let T' be the ET associated with $\mathcal{P}'$. If T contains n buggy nodes, then T' contains n' buggy nodes with $n \leq n'$.*

Proof. Trivial from Theorems 5.4.2 and 5.4.3. On the one hand, equality is ensured with Theorem 5.4.2 because for each buggy node in T, there is at least one buggy node in T'. On the other hand, if a node in T is associated with a function whose code contains more than one loop that has been expanded, then T' can contain more new buggy nodes not present in T. $\square$

5.4.5 Related Work

Reducing the number of questions asked by algorithmic debuggers is a well-known objective in the field, and there exist several works devoted to achieve this goal. Some of them face the problem by defining different ET transformations that modify the structure of the ET to explore it more efficiently.

For instance, the authors of [90] improve the ET produced from Maude programs by introducing nodes. These nodes represent transitivity inferences done by their inference system. Although these nodes could be omitted, if they are kept, the ET becomes more balanced. Balanced ETs are very convenient for search strategies such as Divide and Query [97], because it is possible to prune almost half of the tree after every answer, thus obtaining a logarithmic number of questions with respect to the number of nodes in the ET. This approach is related to our technique, but it has some drawbacks: it can only be applied where transitivity inferences took place while creating the ET, and thus most of the parts of the tree cannot be balanced, and even in these cases the balancing only affects two nodes. Our techniques, in contrast, balance loops, that usually contain many nodes.

Another related approach was presented in [78]. Here, authors introduced a source code (instead of an ET) transformation for list comprehensions in functional programs. Concretely, this technique transforms list comprehensions into a set of equivalent methods that implement the iteration. Even though this technique is used in other paradigm and only works for a different program construct (list comprehensions instead of loops), it is very similar to our loop expansion technique because it transforms the program to implement the list comprehension iterations with recursive functions. This is somehow equivalent to our transformation of `for-each` loops. However, the objective of their technique is different. Their objective is to divide a question related to a list comprehension in different (probably easier) questions, while our objective is to balance the tree, and thus they are optimized in a different way. Of course, their transformation is orthogonal to our technique and it can be applied before.

Even though the techniques discussed can be applied to any language, they only focus on recursion. This means that they cannot improve ETs that use loops, avoiding their use in the imperative or the object-oriented paradigm where loops predominate. Our technique is based on an automatic transformation of loops into recursive methods. Hence, it allows the previously discussed transformations to work in presence of iteration.

5.5 Tree Compression

5.5.1 Introduction

The technique [56] of Algorithmic Debugging (AD) shown in this section is based on the Tree Compression (TC) technique introduced by Davie and Chitil [24]. In particular, we define an algorithm to decide when TC must be applied (or partially applied) in an Execution Tree (ET). TC is a conservative approach that transforms an ET into an equivalent (smaller) ET where we can detect the same bugs. The objective of this technique is to reduce the size of the ET by removing redundant nodes, and it is only applicable to recursive calls. For each recursive call, TC removes the child node associated with the recursive call and all its children become children of the parent node. Let us explain it with an example.

Example 5.5.1 *Consider the ET in Figure 5.21. Here, TC removes six nodes, thus statically reducing the size of the tree. The number by each node represents the number of questions that a particular search strategy (in this case D&Q) needs to find a bug in that node when it is buggy. Observe that the average number of questions has been reduced ($\frac{72}{17}$ vs $\frac{42}{11}$) thanks to the use of TC.*

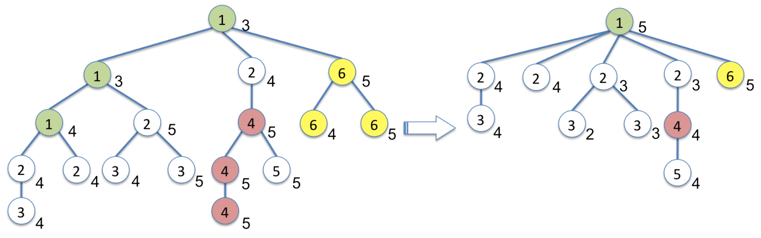


Figure 5.21: Example of Tree Compression

Unfortunately, TC does not always produce good results. Sometimes reducing the number of nodes causes a worse ET structure that is more difficult to debug and thus the number of questions is increased, producing the contrary effect to the intended one.

Example 5.5.2 *Consider the ET in Figure 5.22 (top). In this ET, the average number of questions needed to find the bug is $\frac{33}{9}$. Nevertheless, after compressing the recursive calls (the dark nodes), the average number of questions is augmented up to $\frac{28}{7}$ (see the ET on the left). The reason is that in the new compressed ET we cannot prune any node because its structure is completely flat. The previous structure allowed for pruning some nodes because deep trees are more convenient for AD. However, if we only compress one of the two recursive calls, the number of questions is reduced down to $\frac{27}{8}$ (see the ET on the right).*

Example 5.5.2 clearly shows that TC should not be always applied. To the best of our knowledge, there does not exist an algorithm to decide when to apply TC, and current implementations always compress all recursive calls [23, 46]. Our new technique solves this problem with an analysis to decide when to compress them.

The rest of the section has been organized as follows. Section 5.5.2 introduces some preliminary definitions that will be used in the rest of the section. In Section 5.5.3 we explain our technique and its main applications, and we introduce the algorithm that improves the structure of the ET.

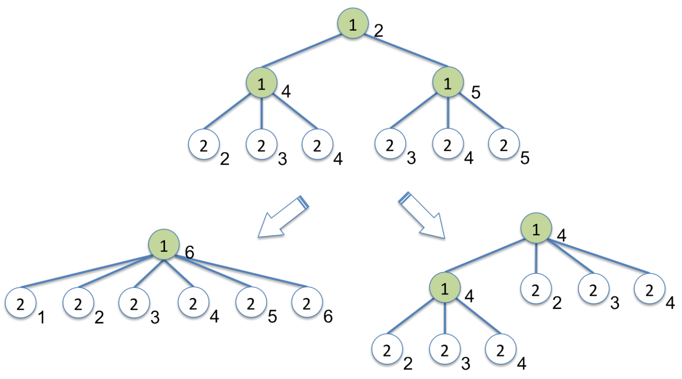


Figure 5.22: Negative and positive effects of Tree Compression

5.5.2 Preliminaries

We use numbers as method identifiers that uniquely identify each method in the source code. This simplification is enough to keep our definitions and algorithms precise and simple. Given an ET, the identifier of the method associated with a node n is referenced with $l(n)$, *simple recursion* is represented with a branch of chained nodes with the same identifier. *Nested recursion* happens when a recursive branch is descendant of another recursive branch. *Multiple recursion* happens when a node labelled with an identifier n has two or more children labelled with n.

In the following, we will refer to the two most used search strategies for AD: Top-Down (TD) [63] and Divide and Query (D&Q) [92]. In both cases, we will always implicitly refer to the most efficient version of both search strategies, respectively named, (i) *Heaviest First* [5], which always traverses the ET from the root to the leaves selecting always the heaviest node; and (ii) *Hirunkitti's Divide and Query* [44], which always selects the node in the ET that better divides the number of nodes in the ET in half. A comparative study of these techniques can be found in [97].

For the comparison of search strategies we use the $\mathcal{Q}uestions(T, s)$ function that computes the number of questions needed (as an average) to find the bug in an ET T using the search strategy s.

5.5.3 Tree Compression Optimization

Tree compression was proposed as a general technique for AD. However, it was defined in the context of a functional language (Haskell) and with the use of a particular search strategy (Hat-Delta). The own authors realized that TC can produce wide trees that are difficult to debug and, for this reason, they defined search strategies that avoid asking repeatedly about the same method. These search strategies do not prevent to apply TC. They just assume that the ET has been totally compressed and they follow a top-down traversal of the ET that can jump to any node when they have a high probability of containing a bug. This way of proceeding somehow partially mitigates the bad structure of the produced ET when it is totally compressed. Our approach is radically different: We do not create a new search strategy to avoid the bad ET structure; but we transform the ET to ensure a good structure.

Even though TC can produce bad ETs (as shown in Example 5.5.2), its authors did not study how this technique works with other (more extended) search strategies such as Top-Down (TD) or Divide & Query (D&Q). So it is not clear at all when to use it. To study when to use TC, we can consider the most general case of a simple recursion in an ET. It is shown in Figure 5.23 where clouds represent possibly empty sets of subtrees and the dark nodes are the recursion branch with a length of $n \geq 2$ calls.

It should be clear that the recursion branch can be useful to prune nodes of the tree. For instance, in the figure, if we ask about the $n/2$ node of the recursion branch, we prune $n/2$ subtrees. Therefore, in the case that the subtrees $T_i, 1 \leq i \leq n$, are empty, then no pruning is possible. In that case, only

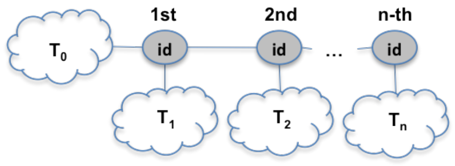


Figure 5.23: Recursion branch in an ET

the nodes in the recursion branch could be buggy; but, because they form a recursive chain, all of them have the same label. Thus, no matter which one is buggy because all of them refer to the same method. Hence, TC must be used to reduce the recursive branch to a single node avoiding search strategies to explore this branch. Therefore, we can conclude that *we must compress every node whose only child is a recursive call.* This result can be formally stated for Top-Down as follows.

Theorem 5.5.3 *Let T be an ET with a recursion branch $R = n_1 \to n_2 \to \ldots \to n_m$ where the only child of a node n_i, $1 \leq i \leq m-1$, is n_{i+1}. And let T' be an ET equivalent to T except that nodes n_i and n_{i+1} have been compressed. Then, $Questions(T', Top\text{-}Down) < Questions(T, Top\text{-}Down)$.*

Proof. Let us consider the two nodes that form the sub-branch to be compressed. For the proof we can call them $n_1 \to n_2$. Firstly, the number of questions needed to find a bug in any ancestor of n_1 is exactly the same if we compress or not the sub-branch. Therefore, it is enough to prove that $Questions(T_{1c},\text{Top-Down}) < Questions(T_1,\text{Top-Down})$ where T_1 is the subtree whose root is n_1 and T_{1c} is the subtree whose root is n_1 after tree compression.
Let us assume that n_2 has j children. Thus we call T_2 the subtree whose root is n_2, and T_{2_i} the subtree whose root is the i-th child of n_2. Then,

$$Questions(T_2,\text{Top-Down}) = \frac{(j+1)+\sum_{i=1}^{j} |T_{2_i}|*(i+Questions(T_{2_i},\text{Top-Down}))}{|T_2|}$$

Here, $(j+1)$ are the questions needed to find a bug in n_2. To reach the children of n_2, the own n_2 and the previous $i-1$ children must be asked about first, and this is why we need to add i to $Questions(T_{2_i},\text{Top-Down})$, Finally, $|T_x|$ represents the number of nodes in the (sub)tree T_x.
Therefore, $Questions(T_{1c},\text{Top-Down}) = Questions(T_2,\text{Top-Down})$
and $Questions(T_1,\text{Top-Down}) = \frac{2+|T_2|*(i+Questions(T_2,\text{Top-Down}))}{|T_2|+1}$
Clearly, $Questions(T_{1c},\text{Top-Down}) < Questions(T_1,\text{Top-Down})$, and thus the claim follows. □

This theorem shows that, in some situations, TC must be used to statically improve the ET structure. But TC is not the panacea, and we need to identify in what cases it should be used. Divide & Query is a good example of a search strategy where TC has a negative effect.

Tree Compression for Divide & Query

In general, when debugging an ET with the D&Q strategy, TC should only be applied in the case described by Theorem 5.5.3 ($T_i, 1 \leq i \leq n$, are empty). The reason is that D&Q can jump to any node of the ET without following a predefined path. This allows D&Q to ask about any node of the recursion branch without asking about the previous nodes in the branch. Note that this does not happen in other search strategies such as Top-Down. Therefore, D&Q has the ability to use the recursion branch as a mean to prune half of the iterations.

Observe in Figure 5.24 that, except for very small recursion branches (e.g., $n \leq 3$), D&Q can take advantage of the recursion branch to prune half of the iterations. The greater n is, the more nodes are pruned. Observe that D&Q can prune nodes even in the case when every node in the recursion branch

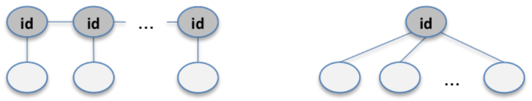


Figure 5.24: Tree Compression applied to a recursive method

only has one child (e.g., $T_i, 1 \leq i \leq n$, are single nodes). Therefore, if we add more nodes to the T_i subtrees, then more nodes can be pruned and D&Q will behave even better.

Tree Compression for Top-Down

In the case of TD-based search strategies, it is not trivial at all to decide when to apply TC. Considering again the ET in Figure 5.23, there are two factors that must be considered: (i) the length n of the recursive branch, and (ii) the size of the trees $T_i, 1 \leq i \leq n$. In order to decide when TC should be used, we provide Algorithm 19 that takes an ET and compresses all recursion branches whenever it improves the ET structure.

Essentially, Algorithm 19 analyzes for each recursion what the effect of applying TC is, and it is finally applied only when it produces an improvement. This analysis is done little by little, separately analyzing each pair of parent-child (recursive) nodes in the sequence. Thus, it is possible that the final result is to only compress one (or several) parts of one recursion branch. For this, the `recs` variable initially contains all nodes of the ET with a recursive child. Each of these nodes is processed with the loop in line 1 in a bottom-up way (lines 2-3). That is, the nodes closer to the leaves are processed first. In order to also consider multiple recursion, the algorithm uses the loops in lines 5 and 8. These loops store in the `improvement` variable the improvement achieved when compressing each recursive branch. In addition to the (`Cost` and `Compress`) functions shown here, the algorithm uses three more functions whose code has not been included because they are trivial: the `Children` function computes the set of children of a node in the ET (i.e., `Children(m) = { n | (m → n) ∈ E }`); the `Sort` function takes a set of nodes and produces an ordered sequence where nodes have been decreasingly ordered by their weights; and the `Pos` function takes a node and a sequence of nodes and returns the position of the node in the sequence.

Given two nodes `parent` and `child` candidates to perform a TC, the algorithm first sorts the children of both the `parent` and the `child` (lines 9-10) in the order in which TD would ask about them (sorted by their weight). Then, it combines the children of both nodes simulating a TC (line 11). Finally, it compares the average number of questions when compressing or not the nodes (line 12). The equation that appears in line 12 is one of the main contributions of the algorithm, because this equation determines when to perform TC between two nodes in a branch with the TD strategy. This equation depends in turn on the formula (line 22 in the Cost function) used to compute the average cost of exploring an ET with TD.

If we analyze Algorithm 19, we can easily realize that its asymptotic cost is quadratic with the number of recursive calls $\mathcal{O}(N^2)$ because in the worst case, all recursive calls would be compared between them. Note also that the algorithm could be used with incomplete ETs [47] (this is useful when we try to debug a program while the ET is being generated, see Section 5.1). In this case, the algorithm can still be applied locally, i.e., to those subtrees of the ET that are totally generated.

Algorithm 19 Optimized Tree Compression

```
Input: An ET $T = (N, E)$
Output: An ET $T'$
Inicialization: $T' = T$ and $recs = \{n \mid n, n' \in N \wedge (n \to n') \in E \wedge l(n) = l(n')\}$

begin
 1) while ($recs \neq \emptyset$)
 2)    take $n \in recs$ such that $\nexists n' \in recs$ with $(n \to n') \in E^+$
 3)    $recs = recs \backslash \{n\}$
 4)    $parent = n$
 5)    do
 6)       $maxImprovement = 0$
 7)       $children = \{c \mid (n \to c) \in E \wedge l(n) = l(c)\}$
 8)       for each $child \in children$
 9)          $pchildren = Sort(Children(parent))$
10)          $cchildren = Sort(Children(child))$
11)          $comb = Sort((pchildren \cup cchildren) \backslash \{child\})$
12)          $improvement = \frac{Cost(pchildren)+Cost(cchildren)}{w_{parent}} - \frac{Cost(comb)}{w_{parent}-1}$
13)          if ($improvement > maxImprovement$)
14)             $maxImprovement = improvement$
15)             $bestNode = child$
16)       end for each
17)       if ($maxImprovement \neq 0$)
18)          $T' = Compress(T', parent, bestNode)$
19)    while ($maxImprovement \neq 0$)
20) end while
21) return $T'$
end

function $Cost(sequence)$
begin
22) return $\sum \{Pos(node, sequence) * w_{node} \mid node \in sequence\} + |sequence|$
end

function $Compress(T = (N, E), parent, child)$
begin
23) $nodes = Children(child)$
24) $E' = E \backslash \{(child \to n) \in E \mid n \in nodes\}$
25) $E' = E' \cup \{(parent \to n) \mid n \in nodes\}$
26) $N' = N \backslash \{child\}$
27) return $T' = (N', E')$
end
```

5.6 Combining Loop Expansion and Tree Compression

We have implemented the original TC algorithm and the optimized version presented in this thesis (see Section 5.5); and also the LE algorithm (see Section 5.4) in such a way that they all can work together. This implementation has been integrated into the Declarative Debugger for Java DDJ [46]. The experiments, the source code of the tool, the benchmarks, and other materials can be found at:

`http://www.dsic.upv.es/~jsilva/DDJ/`

All the implementation has been done in Java. The optimized TC algorithm contains around 90 LOC, and the LE algorithm contains around 1700 LOC. We conducted a series of experiments in order to measure the influence of both techniques in the performance of the debugger. Table 5.10 summarizes the obtained results.

Benchmark	Nodes				LE	Time		Questions				%	
	ET	LE	TC_{ori}	TC_{opt}		LE	TC	ET	LE	TC_{ori}	TC_{opt}	LETC	TC
Factoricer	55	331	51	51	5	151	105	11.62	8.50	7.35	7.35	63.25	100.0
Classifier	25	57	22	24	3	184	4	8.64	6.19	6.46	6.29	72.80	97.36
LegendGame	87	243	87	87	10	259	31	12.81	8.28	11.84	11.84	92.43	100.0
Romanic	121	171	112	113	3	191	12	16.24	7.74	10.75	9.42	58.00	87.62
FibRecursive	5378	6192	98	101	12	251	953	15.64	12.91	9.21	8.00	51.15	86.86
FactTrans	197	212	24	26	3	181	26	10.75	7.88	6.42	5.08	47.26	79.13
BinaryArrays	141	203	100	100	5	172	79	12.17	7.76	7.89	7.89	64.83	100.0
FibFactAna	178	261	44	49	7	202	33	7.90	8.29	8.50	6.06	76.71	71.29
RegresionTest	13	121	15	15	5	237	4	4.77	7.17	4.20	4.20	88.05	100.0
BoubleFibArrays	16	164	10	10	10	213	27	9.31	8.79	4.90	4.90	52.63	100.0
StatsMeanFib	19	50	23	23	6	195	21	7.79	8.12	6.78	6.48	83.18	95.58
Integral	5	8	8	8	3	152	2	6.80	5.75	7.88	5.88	86.47	74.62
TestMath	3	5	3	3	3	195	2	7.67	6.00	9.00	7.67	100.0	85.22
TestMath2	92	2493	13	13	3	211	607	14.70	11.54	15.77	12.77	86.87	80.98
Figures	2	10	10	10	24	597	13	9.00	7.20	6.60	6.60	73.33	100.0
FactCalc	128	179	75	75	3	206	46	8.45	7.60	7.96	7.96	94.20	100.0
SpaceLimits	95	133	98	100	15	786	10	36.26	12.29	18.46	14.04	38.72	76.06

Table 5.10: Experiments applying Loop Expansion and Tree Compression

The first column in Table 5.10 shows the name of the benchmarks. For each benchmark, the `nodes` column shows the number of nodes that are descendant of a loop[11] in the original ET (`ET`), in the ET after applying LE (`LE`), in the ET after applying LE first and then the original version of TC—compressing all nodes—($\texttt{TC}_{ori}$), and in the ET after applying LE first and then the optimized version of TC—Algorithm 19—($\texttt{TC}_{opt}$); the `LE` column shows the number of expanded loops; the `Time` column shows the time (in milliseconds) needed to apply LE and TE; the `Questions` columns show the average number of questions asked, using each of the previously described ETs. Each benchmark has been analyzed assuming that the bug could be in any node of its associated ET. This means that each value in the `Questions` column represents the average of a set of experiments. For instance, in order to obtain the information associated with `Factoricer`, this benchmark has been debugged 55 times with the original ET, 331 with the ET after applying loop expansion, etc. In total, `Factoricer` was debugged 55+331+51+51=488 times, considering all ET transformations and assuming each time that the bug was a different node (and computing the average of all tests for each ET); finally, the `(%)` column shows, on the one hand, the percentage of asked questions after applying our transformations (LE and TC) with respect to the original ET (`LETC`); and, on the other hand, the percentage of questions asked using Algorithm 19 to decide when to apply TC with respect to always applying TC (`TC`). From the table

[11] We consider these nodes because the part of the ET that is not descendant of a loop remains unchanged after applying our technique, and thus the number of questions needed to find the bug is the same before and after the transformations.

we can conclude that LE has a temporal cost of 274 ms whereas TC needs about 123 ms, and that our transformations produce a reduction of 27.65 % in the number of questions asked by the debugger. Moreover, the use of Algorithm 19 to decide when to apply TC also produces an important reduction in the number of questions with an average of 9.72 %.

Chapter 6

Algorithmic Debugging Search Strategies

Divide & Query has been considered the best search strategy for more than 30 years. In this chapter we provide two search strategies that are better than Divide & Query with respect to the number of questions performed by the debugger. First of all we provide a definition of Divide & Query and the reasons why it is not an optimal strategy, later we provide our two new search strategies.

6.1 Divide & Query

6.1.1 Shapiro vs Hirunkitty

In this section we recall the D&Q strategy to show the differences between the original version by Shapiro [92] and the improved version by Hirunkitti and Hogger [44].

Both D&Q by Shapiro and D&Q by Hirunkitti assume that the individual weight of a node is always 1. The individual weight of a node stands for the probability of that node to contain a bug. Therefore, given a Marked Execution Tree (MET) $T = (N, E, M)$ (see Definition 4.2.2), the weight of the subtree rooted at node $n \in N$, w_n, is defined recursively as its number of descendants including itself (i.e., $1 + \sum \{w_{n'} \mid (n \to n') \in E\}$).

D&Q tries to simulate a dichotomic search by selecting the node that better divides the MET into two subMETs with a weight as similar as possible. Therefore, given a MET with n nodes, D&Q searches for the node whose weight is closer to $\frac{n}{2}$. The original algorithm by Shapiro always selects:

- the heaviest node n' whose weight is as close as possible to $\frac{n}{2}$ with $w_{n'} \leq \frac{n}{2}$

Hirunkitti and Hogger noted that this is not enough to divide the MET in half and their improved version always selects the node whose weight is closer to $\frac{n}{2}$ between:

- the heaviest node n' whose weight is as close as possible to $\frac{n}{2}$ with $w_{n'} \leq \frac{n}{2}$, or

- the lightest node n' whose weight is as close as possible to $\frac{n}{2}$ with $w_{n'} \geq \frac{n}{2}$

Because it is strictly better, in the rest of the chapter we only consider Hirunkitti's D&Q and refer to it as D&Q.

6.1.2 Limitations of Divide & Query

In this section we show that D&Q is suboptimal when the MET does not contain a wrong node (i.e., all nodes are marked as undefined).[1] The intuition beyond this limitation is that the objective of D&Q is to divide the tree in half, but the real objective should be to reduce the number of questions the oracle would be asked. For instance, consider the MET in Figure 6.1 (left) where the black node is marked as wrong and D&Q would select the grey node. The objective of D&Q is to divide the 8 nodes into two groups of 4. Nevertheless, the real motivation of dividing the tree should be to divide the tree into two parts that would produce the same number of remaining questions (in this case 3).

The problem comes from the fact that D&Q does not take into account the marking of wrong nodes. For instance, observe the two METs in Figure 6.1 (centre) where each node is labelled with its weight and the black node is marked as wrong. In both cases D&Q would behave exactly in the same way, because it completely ignores the marking of the root. Nevertheless, it is evident that we do not need to ask again about a node that is already marked as wrong to determine whether it is buggy. However, D&Q counts the nodes marked as wrong as part of their own weight, and this is a source of inefficiency.

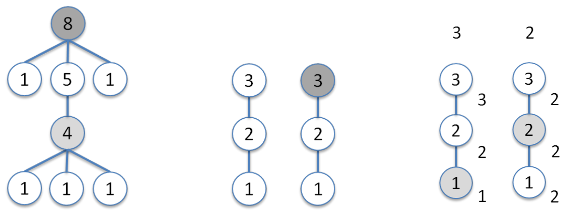


Figure 6.1: Behaviour of Divide and Query

In the METs of Figure 6.1 (centre) we have two METs. In the one on the right, the nodes with weight 1 and 2 are optimal, but in the one on the left, only the node with weight 2 is optimal. In both METs D&Q would select either the node with weight 1 or the node with weight 2 (both are equally close to $\frac{3}{2}$). However, we show in Figure 6.1 (right) that selecting the node with weight 1 is suboptimal, and the search strategy should always select the node with weight 2. Considering that the grey node is the first node selected by the search strategy, then the number on the side of a node represents the number of questions needed to find the bug if the buggy node is this node. The number at the top of the figure represents the number of questions needed to determine that there is not a bug. Clearly, as an average, it is better to select first the node with weight 2 because we would perform less questions ($\frac{8}{4}$ vs. $\frac{9}{4}$ considering all four possible cases).

Therefore, D&Q returns a set of nodes that contains the best node, but it is not able to determine which of them is the best node, thus being suboptimal when it is not selected. In addition, the METs in Figure 6.2 show that D&Q is incomplete. Observe that the METs have 5 nodes, thus D&Q would always select the node with weight 2. However, the node with weight 4 is equally optimal (both need $\frac{16}{6}$ questions as an average to find the bug) but it will be never selected by D&Q because its weight is far from the half of the tree $\frac{5}{2}$.

Another limitation of D&Q is that it was designed to work with METs where all the nodes have the same individual weight, and moreover, this weight is assumed to be one. This is because D&Q assumes that all the nodes have the same probability of containing the bug, and thus an individual weight of one for all nodes is enough to represent this. If we work with METs where nodes can have different individual weights (i.e., the nodes have different probabilities of containing the bug) and these weights can be any value greater or equal to zero, then D&Q is suboptimal as it is demonstrated by the MET in Figure 6.3. In this MET, D&Q would select the n_1 node because its weight is closer to $\frac{21}{2}$ than any

[1]Modern debuggers [46] allow the user to debug the MET while it is being generated (see Section 5.1). Thus the root node of the subtree being debugged is not necessarily marked as *Wrong*.

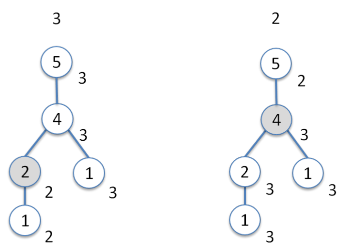


Figure 6.2: Incompleteness of Divide and Query

other node. However, the n_2 node is the node that better divides the tree in two parts with the same probability of containing the bug.

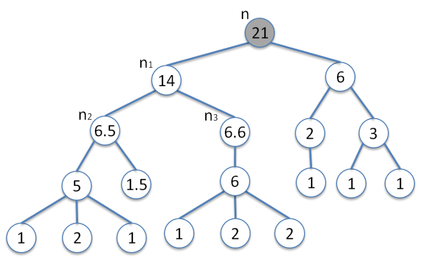


Figure 6.3: MET with decimal individual weights

In summary,

1. D&Q is suboptimal when the MET is free of wrong nodes,

2. D&Q is correct when the MET contains wrong nodes and all the nodes of the MET have the same weight, but

3. D&Q is suboptimal when the MET contains wrong nodes and the nodes of the MET have different individual weights.

Despite in this section we have shown the problems D&Q has, those problems only show that the strategy does not properly perform its job, but we have never said that D&Q is an optimal search strategy or not. In the next section we claim and prove that D&Q based strategies are suboptimal.

6.1.3 Divide & Query is not an optimal search strategy

In this section we show that the D&Q strategy is not optimal. The reason is that the initial hypothesis of D&Q, which says to divide the tree in two subtrees with the same number of nodes, does not actually convert the search strategy into a dichotomic search.

An AD search strategy is optimal if the average number of questions that it performs in any MET is minimum. We can calculate the number of performed questions assuming that the bug can be located in the code associated with any node of the tree and, therefore, calculating the sequences of questions that the debugger would ask for each node. Evidently, there can exist several different optimal strategies. From here on, we will call *optimal node* to the first node that an optimal search strategy would ask about.

Definition 6.1.1 (Optimal search strategy) *Let ϵ be an AD search strategy. Given a MET $T = (N, E, M)$, being s_n^ϵ the sequence of questions performed by Algorithm 1 using the ϵ search strategy, and* *assuming that the only buggy node of T is $n \in N$. Let $t_\epsilon = \sum_{n_i \in N} |s^\epsilon_{n_i}|$. We say that ϵ is an optimal search strategy if for any MET we have that $\nexists \epsilon'.t_\epsilon > t_{\epsilon'}$.*

We show here three counterexamples where we can see that the information D&Q uses to select the optimal node is not enough. The examples are based on the cost (measured in number of questions performed by the debugger) associated with the nodes selected by D&Q. To measure this cost, we use the following definition of sequence of questions.

Definition 6.1.2 (Sequence of questions) *Given a MET $T = (N, E, M)$ and two nodes $n_1, n_2 \in N$, the* sequence of questions *of n_1 with respect to n_2, $sp(n_1, n_2)$, is formed by all the questions performed by Algorithm 1 assuming that the first node selected by selectNode(T) function is n_2 and that n_1 is the only buggy node in T.*

Intuitively, $sp(n_1, n_2)$ is composed of the sequence of questions that the debugger would ask to determine that n_1 is a buggy node assuming that the first question of the sequence is associated with n_2. This implies that the sequence of questions completely depends on the search strategy used. In the concrete case of D&Q, in the MET of Figure 6.4 (left) we can see that:

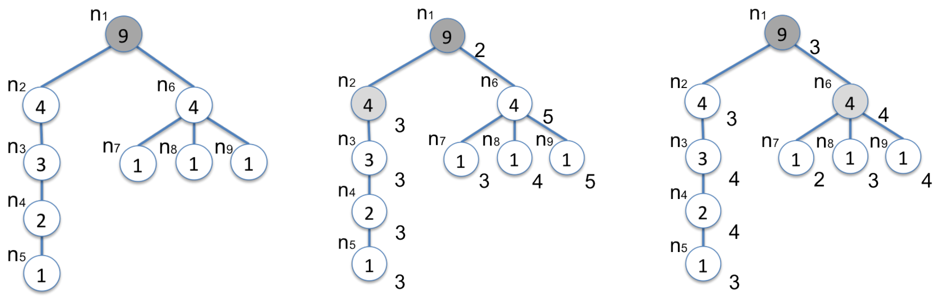


Figure 6.4: Divide & Query counterexample

$$sp(n_2, n_6) = [n_6, n_3, n_2] \qquad sp(n_6, n_2) = [n_2, n_6, n_7, n_8, n_9]$$
$$sp(n_2, n_2) = [n_2, n_4, n_3] \qquad sp(n_6, n_6) = [n_6, n_7, n_8, n_9]$$

In the middle and right METs of Figure 6.4, as well as in the following figures, the number in the bottom right corner of a node represents the size of the sequence of questions of this node with respect to the grey node. This is, the number of questions that should be asked to find an error assuming that it will be found at that node and that the first question is associated with the grey node.

Example 6.1.3 *Consider the MET of Figure 6.4 (left). In this figure, the root node is marked as wrong and D&Q selects n_2 and n_6 as optimal nodes. When adding up the number of questions performed in the MET of Figure 6.4 (centre), where the first question is associated with the n_2 node, we can see that the total number of questions is 31, producing an average of $\frac{31}{9} = 3,44$ questions to find the bug wherever it is in the 9 nodes of the tree. However, in the MET of Figure 6.4 (right), in which the first question is associated with the n_6 node, the total number of questions is 30, hence having an average of $\frac{30}{9} = 3,33$ questions.*

It can be seen that, despite what D&Q assumes, starting selecting the node that divides the tree in two subtrees with the same quantity of nodes is not enough to determine which is the optimal node. In this example both nodes have the same weight (4) and both divide the tree in a similar way. But we have shown that starting asking about the n_6 node produces less questions on average than starting asking about the n_2 node. Being, hence, the n_6 node optimal instead of n_2. To explain the reason, we use the following example.

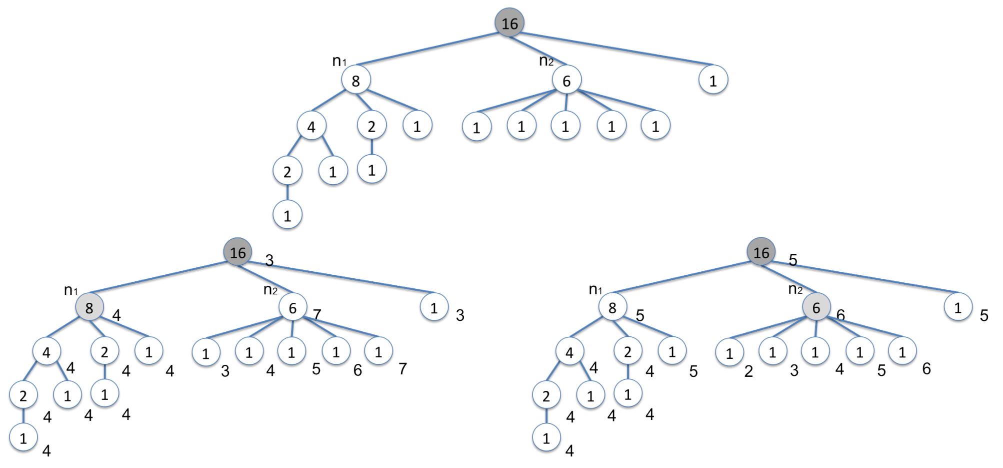


Figure 6.5: Divide & Query second counterexample

Example 6.1.4 *In the MET at the top of Figure 6.5 we can observe that the subtree whose root is n_1 is a completely balanced (sub)MET [47] (i.e., in the worst case the subtree can be debugged in a logarithmic number of questions). But the subtree whose root is n_2 is a completely unbalanced (sub)MET (i.e., in the worst case the debugger would ask about all the nodes to find the bug).*

D&Q tries to do a logarithmical search by asking about the node that divides the weight in half, in this case $\frac{16}{2} = 8$, hence the n_1 node is optimal according to D&Q. In the MET of Figure 6.5 (left) the numbers of questions performed when starting to ask about this node adds up 70 questions in total, this is $\frac{70}{16} = 4,38$ as an average to find the bug wherever it is in the 16 nodes. However, if we start asking about the n_2 node, the same amount of questions are performed. The numbers of questions of this last case can be seen in the MET of Figure 6.5 (right). Even though both nodes are equally good, the n_2 node will never be selected by D&Q because its weight is farther away from $\frac{16}{2}$ than n_1.

The reason why the n_2 node is equally good as n_1 is straightforward. Having a look at the top tree of Figure 6.5 we can see that, if n_2 is the first node to be asked about and the bug is not in its subtree, then the remaining subtree can be easily explored thanks to its balanced structure. However, in the other option this is not true. If n_1 is the first node to be asked about and the bug is not in its subtree, then the remaining subtree is completely unbalanced and thus it would be very difficult to explore. This can be seen comparing the left tree with the right tree of Figure 6.5: almost all nodes of the subtree whose root is n_1 do not increment their number of questions regardless of which the first node to be asked about is, whereas asking first about n_1 would produce an increment of questions in the nodes of the subtree whose root is n_2.

All this take us to the conclusion that it is not only important to prune big subtrees of the MET, but it is also important to prune unbalanced trees that are more difficult to explore. The latter is ignored by D&Q and this is the reason why it is not an optimal search strategy.

Finally, we show an example in which D&Q is not only unable to find all optimal nodes as in the previous case, but it cannot even find any of them. The example is the same tree from Example 6.1.3 in which we have added 59 nodes in a deep way to the n_2 node and 46 nodes in a wide way to the n_6 node.

Example 6.1.5 *Consider the MET of Figure 6.6 (left) in which D&Q determines that the optimal node should be in the left branch of the tree, concretely the n_1 node because it is the node whose weight is as close as possible to $\frac{114}{2} = 57$. However, the right child n_2 is the optimal node because it produces less questions than n_1, in spite of having a weight farther from 57.*

In the tree in the middle of Figure 6.6 the first question is associated with the node selected by D&Q, n_1. If the bug were in the subtree whose root is n_1, because the tree is a sequence of nodes in depth, the

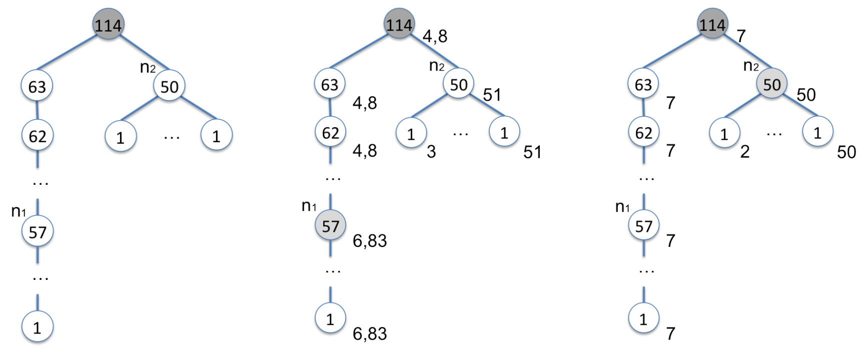


Figure 6.6: Divide & Query third counterexample

search strategy would ask $\log_2 57 = 5,83$ *questions as an average to each node (*$+1$ *because of having initially asked about the* n_1 *node). If the error were not in the subtree whose root is* n_1 *after asking about it, the subtree whose root is the* n_2 *node would start to be explored. If the error is in this subtree, the search strategy would ask about all nodes until it finally finds it, and all of them carry the question previously asked about the* n_1 *node, in total it is* $(\sum_{i=3}^{51} i) + 51 = 1374$ *questions. If finally it is not in that subtree either, then the bug must be in the rest 7 nodes above the* n_1 *node (including the root). In such nodes the search strategy would perform* $\log_2 7 = 2,8$ *questions as an average to each node (*$+2$ *because of asking about* n_1 *and* n_2 *nodes). In total it sums* $57 * 6,83 + 1374 + 7 * 4,8 = 1796,91$ *questions, and an average of* $\frac{1796,91}{114} = 15,76$ *questions.*

Contrarily, if we start by asking directly about n_2 *and the bug is in this subtree, then the search strategy would make* $(\sum_{i=2}^{50} i) + 50 = 1324$ *because they do not have any carried questions. If the bug is in the other branch, after asking about the* n_2 *node there are 64 nodes left in depth, so in* $\log_2 64 = 6$ *questions (*$+1$ *because of having previously asked about the* n_2 *node) the error would be found. In total it sums* $1324 + 64 * 7 = 1772$ *questions, and an average of* $\frac{1772}{114} = 15,54$ *questions.*

These counterexamples confirm that D&Q is not always able to find all optimal nodes, being then not complete. Moreover, it also shows that in some cases, D&Q selects a node that it is not optimal, being then not correct either. Additionally, the examples have revealed that besides considering the amount of nodes, the structure of the tree should also be consider to obtain a search strategy that performs less questions as possible to the oracle.

6.2 Optimal Divide & Query

6.2.1 Introduction

The search strategy used to decide which nodes of the Marked Execution Tree (MET) should be asked about is crucial for the performance of Algorithmic Debugging (AD). Since the definition of AD, there have been a lot of research concerning the definition of new search strategies trying to minimize the number of questions [95, 97]. We conducted several experiments to measure the performance of all current AD search strategies. The results of the experiments are shown in Figure 6.7, where the first column contains the names of the benchmarks; the `nodes` column shows the number of nodes in the MET associated with each benchmark; and the other columns represent AD search strategies [95] that are ordered according to their performance: Optimal Divide & Query (`D&QO`), Divide & Query by Hirunkitti (`D&QH`), Divide & Query by Shapiro (`D&QS`), Divide by Rules & Query (`DR&Q`), Heaviest First (`HF`), More Rules First (`MRF`), Hat Delta Proportion (`HD-P`), Top-Down (`TD`), Hat Delta YES (`HD-Y`), Hat Delta NO (`HD-N`), Single Stepping (`SS`).

Benchmark	Nodes	D&QO	D&QH	D&QS	DR&Q	HF	MRF	HD-P	HD-Y	TD	HD-N	SS	Average
NumReader	*12*	28,99	28,99	31,36	29,59	44,38	44,38	49,70	49,70	49,70	49,70	53,25	41,80
Orderings	*46*	12,04	12,09	12,63	14,40	17,16	17,29	21,05	20,60	20,82	19,60	51,02	19,88
Factoricer	*62*	9,83	9,83	9,93	20,03	12,55	12,55	15,04	15,04	12,55	18,29	50,77	16,94
Sedgewick	*12*	30,77	30,77	33,14	30,77	34,91	34,91	43,79	43,79	43,20	43,79	53,25	38,46
Clasifier	*23*	19,79	20,31	22,40	21,88	22,92	23,26	32,12	32,12	31,94	34,55	51,91	28,47
LegendGame	*71*	8,87	8,87	8,95	16,72	11,15	11,23	14,68	14,68	13,37	16,94	50,68	16,01
Cues	*18*	31,58	32,41	32,41	32,41	33,24	34,63	39,06	39,06	42,11	44,32	52,35	37,60
Romanic	*123*	6,40	10,84	11,23	13,56	7,44	11,88	13,29	13,29	13,41	13,30	50,40	15,00
FibRecursive	*4.619*	0,27	0,27	0,28	1,20	0,33	0,41	3,92	3,92	0,46	0,48	50,01	5,59
Risk	*33*	16,78	16,78	18,08	19,38	18,69	18,69	24,31	24,31	31,14	32,79	51,38	24,76
FactTrans	*198*	3,89	3,89	3,93	6,22	6,58	6,58	7,37	7,24	7,16	7,50	50,25	10,06
RndQuicksort	*72*	8,73	8,73	8,73	11,41	12,03	12,23	13,62	12,93	13,51	14,54	50,67	15,19
BinaryArrays	*128*	5,52	5,52	5,71	7,13	7,75	7,94	7,90	8,15	8,59	8,71	50,38	11,21
FibFactAna	*351*	2,44	2,44	2,45	5,38	7,61	7,71	6,40	7,39	8,57	5,99	50,14	9,68
NewtonPol	*7*	39,06	39,06	43,75	39,06	43,75	43,75	45,31	45,31	45,31	45,31	54,69	44,03
RegresionTest	*18*	23,27	23,27	25,21	25,21	26,87	26,87	32,96	32,96	32,96	32,96	52,35	30,45
BoubleFibArrays	*171*	4,40	4,41	4,57	11,40	5,95	6,96	24,50	24,87	6,96	6,96	50,29	13,75
ComplexNumbers	*60*	10,02	10,02	10,32	11,31	11,39	11,39	15,78	15,80	15,75	19,19	50,79	16,53
Integral	*5*	44,44	44,44	47,22	44,44	50,00	50,00	50,00	50,00	50,00	50,00	55,56	48,74
TestMath	*48*	11,91	11,91	12,16	12,99	15,95	16,28	22,41	23,87	24,20	22,37	50,98	20,46
TestMath2	*228*	3,51	3,51	3,51	9,73	10,55	10,81	12,29	13,24	28,56	14,37	50,22	14,57
Figures	*113*	6,72	6,75	6,79	8,09	7,68	7,79	10,17	10,16	10,60	10,76	50,43	12,36
FactCalc	*59*	10,11	10,14	10,42	11,53	13,69	14,22	20,47	20,47	18,50	20,69	50,81	18,28
SpaceLimits	*127*	12,95	16,07	19,15	21,74	13,68	16,80	22,87	22,86	22,78	26,15	50,38	22,31
Average	**275,17**	**14,68**	**15,06**	**16,01**	**17,73**	**18,18**	**18,69**	**22,87**	**22,99**	**23,01**	**23,30**	**51,37**	**22,17**

Figure 6.7: Performance of AD search strategies

For each benchmark, we produced its associated MET and assumed that the buggy node could be any node of the MET (i.e., any subcomputation in the execution of the program could be buggy). Therefore, we performed a different experiment for each possible case and, hence, each cell of the table summarizes a number of experiments that were automatized. In particular, the *Factoricer* benchmark has been debugged 62 times with each search strategy; each time we selected a different node and simulated that it was buggy, thus the results shown are the average number of questions performed by each search strategy with respect to the number of nodes (i.e., the mean percentage of asked nodes). Similarly, the *Cglib* benchmark has been debugged 1216 times with each search strategy, and so on.

Observe that the best AD search strategies in practice are the two variants of Divide and Query (ignoring our new technique D&QO that reduces the average number of questions by 0.5 points (see Figure 6.7). Moreover, from a theoretical point of view, this search strategy has been thought optimal in the worst case, and it has been implemented in almost all current algorithmic debuggers (see, e.g., [13, 87, 24, 46, 85]). Moreover, the original search strategy was only defined for METs where all the nodes have an individual weight of 1. In contrast, we allow our algorithms to work with different individual weights that can be integer, but also decimal. An individual weight of zero means that this node cannot contain the bug. A positive individual weight approximates the probability of being buggy. The higher the individual weight is, the higher the probability is. This generalization strongly influences the technique and allows for assigning different probabilities of being buggy to different parts of the program. For instance, a recursive function with higher-order calls should be assigned a higher individual weight than a function implementing a simple base case [95]. The weight of the nodes can also be reassigned dynamically during the debugging session in order to take into account the oracle's answers [24].

We show that the original algorithms are inefficient with METs where nodes can have different individual weights in the domain of the positive real numbers (including zero) and we redefine the technique for these generalized METs.

The rest of the section has been organized as follows. In Section 6.2.2 we introduce some preliminary definitions that will be used in the rest of the section. Section 6.2.3 introduces a new algorithm that is optimal for METs whose nodes have the same individual weights. In Sections 6.2.4 and 6.2.5 we extend the algorithm of Section 6.2.3 to also consider METs whose nodes can have different individual weights including and excluding zero, respectively. Finally, detailed proofs of the results shown in the section are presented in Section 6.2.6.

6.2.2 Preliminary definitions

In the next sections we introduce a new version of D&Q [48, 50] that tries to divide the MET into two parts with the same probability of containing the bug (instead of two parts with the same weight). We introduce new algorithms that are correct and complete even if the MET contains nodes with different individual weights. For this, we firstly need to define the *search area* of a MET, which is the set of undefined nodes.

Definition 6.2.1 (Search area) *Let* $T = (N, E, M)$ *be a MET. The* search area *of* T, $Sea(T)$, *is defined as* $\{\ n \in N \mid M(n) = Undefined\ \}$.

While D&Q uses the whole T, we only use $Sea(T)$, because answering all nodes in $Sea(T)$ guarantees that we can discover all buggy nodes [63]. Moreover, in the following we refer to the individual weight of a node n with wi_n; and we refer to the weight of a (sub)tree rooted at n with w_n that is recursively defined as:

$$w_n = \begin{cases} \sum \{w_{n'} \mid (n \to n') \in E\} & \text{if } M(n) \neq Undefined \\ wi_n + \sum \{w_{n'} \mid (n \to n') \in E\} & \text{otherwise} \end{cases}$$

Note that, contrarily to standard D&Q, the definition of w_n excludes those nodes that are not in the search area (i.e., the root node when it is wrong). Note also that wi_n allows for assigning any individual weight to the nodes. This is an important generalization of D&Q where it is assumed that all nodes have the same individual weight and it is always 1.

6.2.3 Debugging METs where all nodes have the same individual weight

For the sake of clarity, given a node $n \in Sea(T)$, we distinguish between three subareas of $Sea(T)$ induced by n: (1) n itself, whose individual weight is wi_n; (2) descendants of n, whose weight is

$$Down(n) = \sum \{wi_{n'} \mid n' \in Sea(T) \wedge (n \to n') \in E^+\}$$

and (3) the rest of nodes, whose weight is

$$Up(n) = \sum \{wi_{n'} \mid n' \in Sea(T) \wedge (n \to n') \notin E^*\}$$

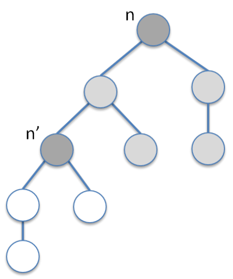


Figure 6.8: Functions Up and Down of Optimal Divide & Query

Example 6.2.2 *Consider the MET in Figure 6.8.*

Assuming that the root n is marked as wrong and all nodes have an individual weight of 1, then $Sea(T)$ contains all nodes except n, $Up(n') = 4$ (total weight of the grey nodes), and $Down(n') = 3$ (total weight of the white nodes).

Clearly, for any MET whose root is n and a node n', $M(n') = Undefined$, we have that:

$$w_n = Up(n') + Down(n') + wi_{n'} \qquad \text{(Equation 1)}$$
$$w_{n'} = Down(n') + wi_{n'} \qquad \text{(Equation 2)}$$

Intuitively, given a node n, what we want to divide in half is the area formed by $Up(n) + Down(n)$. That is, n will not be part of $Sea(T)$ after being answered, thus the objective is to make $Up(n)$ equal to $Down(n)$. This is another important difference with traditional D&Q: wi_n should not be considered when dividing the MET. We use the notation $n_1 \gg n_2$ to express that n_1 divides $Sea(T)$ better than n_2 (i.e., $|Down(n_1) - Up(n_1)| < |Down(n_2) - Up(n_2)|$). And we use $n_1 \equiv n_2$ to express that n_1 and n_2 equally divide $Sea(T)$. If we find a node n such that $Up(n) = Down(n)$ then n produces an optimal division, and should be selected by the search strategy. If an optimal solution cannot be found, the following theorem states how to compare the nodes in order to decide which of them should be selected.

Theorem 6.2.3 *Given a MET $T = (N, E, M)$ whose root is $n \in N$, where $\forall n', n'' \in N, wi_{n'} = wi_{n''}$ and $\forall n' \in N, wi_{n'} > 0$, and given two nodes $n_1, n_2 \in Sea(T)$, with $w_{n_1} > w_{n_2}$, $n_1 \gg n_2$ if and only if $w_n > w_{n_1} + w_{n_2} - wi_n$.*

Theorem 6.2.4 *Given a MET $T = (N, E, M)$ whose root is $n \in N$, where $\forall n', n'' \in N, wi_{n'} = wi_{n''}$ and $\forall n' \in N, wi_{n'} > 0$, and given two nodes $n_1, n_2 \in Sea(T)$, with $w_{n_1} > w_{n_2}$, $n_1 \equiv n_2$ if and only if $w_n = w_{n_1} + w_{n_2} - wi_n$.*

Theorems 6.2.3 and 6.2.4 are useful when one node is heavier than the other. In the case that both nodes have the same weight, then the following theorem guarantees that they both equally divide the MET in all situations.

Theorem 6.2.5 *Let $T = (N, E, M)$ be a MET where $\forall n, n' \in N, wi_n = wi_{n'}$ and $\forall n \in N, wi_n > 0$, and let $n_1, n_2 \in Sea(T)$ be two nodes, if $w_{n_1} = w_{n_2}$ then $n_1 \equiv n_2$.*

Corollary 6.2.6 *Given a MET $T = (N, E, M)$ where $\forall n, n' \in N, wi_n = wi_{n'}$ and $\forall n \in N, wi_n > 0$, and given a node $n \in Sea(T)$, then n optimally divides $Sea(T)$ if and only if $Up(n) = Down(n)$.*

While Corollary 6.2.6 states the objective of optimal D&Q (finding a node n such that $Up(n) = Down(n)$), Theorems 6.2.3 and 6.2.5 provide a method to approximate this objective (finding a node n such that $|Down(n) - Up(n)|$ is minimum in $Sea(T)$).

An algorithm for Optimal Divide & Query.

Theorems 6.2.3 and 6.2.4 provide equation $w_n \geq w_{n_1} + w_{n_2} - wi_n$ to compare two nodes n_1, n_2 by efficiently determining $n_1 \gg n_2$, $n_1 \equiv n_2$ or $n_1 \ll n_2$. However, with only this equation, we should compare all nodes to select the best of them (i.e., n such that $\nexists n', n' \gg n$). Hence, in this section we provide an algorithm that allows for finding the best node in a MET with a minimum set of node comparisons.

Given a MET, Algorithm 20 efficiently determines the best node to divide $Sea(T)$ in half (in the following the *optimal node*). In order to find this node, the algorithm does not need to compare all nodes in the MET. It follows a path of nodes from the root to the optimal node that is closer to the root producing a minimum set of comparisons.

Algorithm 20 Optimal D&Q —SelectNode in Algorithm 1—

```
Input: A MET $T = (N, E, M)$ whose root is $n \in N$,
       $\forall n', n'' \in N, wi_{n'} = wi_{n''}$ and $\forall n' \in N, wi_{n'} > 0$
Output: A node $n_{Optimal} \in N$
Preconditions: $\exists n' \in N, M(n') = Undefined$

begin
  1) $Candidate = n$
  2) do
  3)    $Best = Candidate$
  4)    $Children = \{m \mid (Best \to m) \in E\}$
  5)    if $(Children = \emptyset)$ then return $Best$
  6)    $Candidate = n' \mid \forall n''$ with $n', n'' \in Children, w_{n'} \geq w_{n''}$
  7) while $(w_{Candidate} > \frac{w_n}{2})$
  8) if $(M(Best) = Wrong)$ then return $Candidate$
  9) if $(w_n \geq w_{Best} + w_{Candidate} - wi_n)$ then return $Best$
 10)                                            else return $Candidate$
end
```

Example 6.2.7 *Consider the MET in Figure 6.9 where $\forall n \in N, wi_n = 1$ and $M(n) = Undefined$. Observe that Algorithm 20 only needs to apply the equation in Theorem 6.2.3 once to identify an optimal*

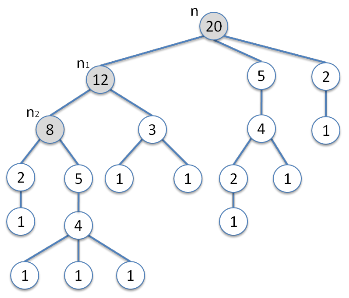


Figure 6.9: Defining a path in a MET to find the optimal node

node. Firstly, it traverses the MET top-down from the root selecting at each level the heaviest node until we find a node whose weight is smaller than the half of the MET ($\frac{w_n}{2}$), thus, defining a path in the MET that is coloured in grey. Then, the algorithm uses the equation $w_n \geq w_{n_1} + w_{n_2} - wi_n$ to compare the n_1 and n_2 nodes. Finally, the algorithm selects n_1.

In order to prove the correctness of Algorithm 20, we need to prove that (1) the node returned is really an optimal node, and (2) this node will always be found by the algorithm (i.e., it is always in the path defined by the algorithm).

The first point can be proved with Theorems 6.2.3, 6.2.4 and 6.2.5. The second point is the key idea of the algorithm and it relies on an interesting property of the defined path: while defining the path in the MET, only four cases are possible, and all of them coincide in that the subtree of the heaviest node will contain an optimal node.

In particular, when we use Algorithm 20 and compare two nodes n_1, n_2 in a MET whose root is n, we find four possible cases:

Case 1: n_1 and n_2 are siblings.
Case 2: $w_{n_1} > w_{n_2} \ \wedge \ w_{n_2} > \frac{w_n}{2}$.
Case 3: $w_{n_1} > \frac{w_n}{2} \ \wedge \ w_{n_2} \leq \frac{w_n}{2}$.
Case 4: $w_{n_1} > w_{n_2} \ \wedge \ w_{n_1} \leq \frac{w_n}{2}$.

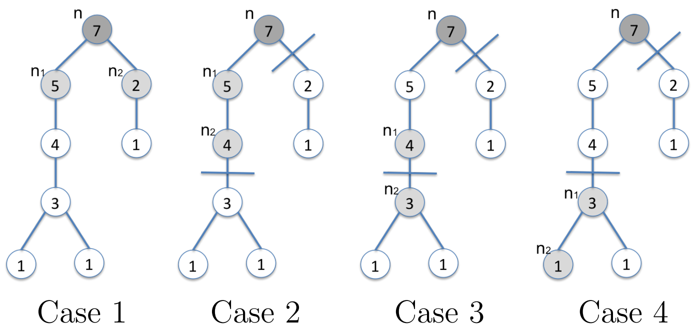


Figure 6.10: Determining the best node in a MET (four possible cases)

We have proved—the individual proofs are part of the proof of Theorem 6.2.8—that in cases 1 and 4, the heaviest node is better (i.e., if $w_{n_1} > w_{n_2}$ then $n_1 \gg n_2$); In case 2, the lightest node is better; and in case 3, the best node must be determined with the equations of Theorems 6.2.3, 6.2.4 and 6.2.5. Observe that these results allow the algorithm to determine the path to the optimal node that is closer to the root. For instance, in Example 6.2.7 case 1 is used to select a child, e.g., node 12 instead of node 5 or node 2, and node 8 instead of node 3. Case 2 is used to go down and select node 12 instead of node 20. Case 4 is used to stop going down at node 8 because it is better than all its descendants. And it is also used to determine that nodes 2, 3 and 5 are better than all their descendants. Finally, case 3 is used to select the optimal node, 12 instead of 8. Note that D&Q could have selected node 8 that is equally close to $\frac{20}{2}$ than node 12; but it is suboptimal because $Up(8) = 12$ and $Down(8) = 7$ whereas $Up(12) = 8$ and $Down(12) = 11$.

The correctness of Algorithm 20 is stated by the following theorem.

Theorem 6.2.8 (Correctness) *Let $T = (N, E, M)$ be a MET where $\forall n, n' \in N, wi_n = wi_{n'}$ and $\forall n \in N, wi_n > 0$, then the execution of Algorithm 20 with T as input always terminates producing as output a node $n \in Sea(T)$ such that $\nexists n' \in Sea(T) \mid n' \gg n$.*

Algorithm 20 always returns a single optimal node. However, the equation in Theorem 6.2.3 in combination with the equation in Theorem 6.2.4 can be used to identify all optimal nodes in the MET. This is implemented in Algorithm 21 that is complete, and thus it returns nodes 2 and 4 in the MET of Figure 6.2 where D&Q can only detect node 2 as optimal.

Algorithm 21 Optimal D&Q (Complete) —SelectNode in Algorithm 1—

```
Input: A MET $T = (N, E, M)$ whose root is $n \in N$,
       $\forall n', n'' \in N, wi_{n'} = wi_{n''}$ and $\forall n' \in N, wi_{n'} > 0$
Output: A set of nodes $O \subseteq N$
Preconditions: $\exists n' \in N, M(n') = Undefined$

begin
  1) $Candidate = n$
  2) do
  3)    $Best = Candidate$
  4)    $Children = \{m \mid (Best \rightarrow m) \in E\}$
  5)    if $(Children = \emptyset)$ then return $\{Best\}$
  6)    $Candidate = n' \mid \forall n''$ with $n', n'' \in Children, w_{n'} \geq w_{n''}$
  7) while $(w_{Candidate} > \frac{w_n}{2})$
  8) $Candidates = \{n' \mid \forall n''$ with $n', n'' \in Children, w_{n'} \geq w_{n''}\}$
  9) if $(M(Best) = Wrong)$ then return $Candidates$
 10) if $(w_n > w_{Best} + w_{Candidate} - wi_n)$ then return $\{Best\}$
 11) if $(w_n = w_{Best} + w_{Candidate} - wi_n)$ then return $\{Best\} \cup Candidates$
 12)                                             else return $Candidates$
end
```

6.2.4 Debugging METs where nodes can have different individual weights (including zero)

In this section we generalize Divide and Query to the case where nodes can have different individual weights and these weights can be any value greater or equal to zero. As shown in Figure 6.3, in this general case traditional D&Q fails to identify the optimal node (it selects node n_1 but the optimal node is n_2). The algorithm presented in the previous section is also suboptimal when the individual weights can be different. For instance, in the MET of Figure 6.3, it would select node n_3. For this reason, in this section we introduce Algorithm 22, a general algorithm able to identify an optimal node in all cases. It does not mean that Algorithm 20 is useless. Algorithm 20 is optimal when all nodes have the same weight, and in that case, it is more efficient than Algorithm 22. Theorem 6.2.9 ensures the finiteness and correctness of Algorithm 22.

Theorem 6.2.9 (Correctness) *Let $T = (N, E, M)$ be a MET where $\forall n \in N, wi_n \geq 0$, then the execution of Algorithm 22 with T as input always terminates producing as output a node $n \in Sea(T)$ such that $\nexists n' \in Sea(T) \mid n' \gg n$.*

6.2.5 Debugging METs where nodes can have different individual weights (excluding zero)

In the previous section we provided an algorithm that optimally selects an optimal node of the MET with a minimum set of node comparisons. But this algorithm is not complete due to the fact that we allow the nodes to have an individual weight of zero. For instance, when all nodes have an individual weight of zero, Algorithm 22 returns a single optimal node, but it is not able to find all optimal nodes.

Given a node (say n), the difference between having an individual weight of zero, wi_n, and having a (total) weight of zero, w_n, should be clear. The former means that this node did not cause a bug, the later means that none of the descendants of this node (including the node itself) caused a bug. Surprisingly, the use of nodes with individual weights of zero has not been exploited in the literature.

Algorithm 22 Optimal D&Q General —SelectNode in Algorithm 1—

```
Input: A MET $T = (N, E, M)$ whose root is $n \in N$ and $\forall n' \in N, wi_{n'} \geq 0$
Output: A node $n_{Optimal} \in N$
Preconditions: $\exists n' \in N, M(n') = Undefined$

begin
 1) $Candidate = n$
 2) do
 3)    $Best = Candidate$
 4)    $Children = \{m \mid (Best \to m) \in E\}$
 5)    if $(Children = \emptyset)$ then return $Best$
 6)    $Candidate = n' \mid \forall n''$ with $n', n'' \in Children, w_{n'} \geq w_{n''}$
 7) while $(w_{Candidate} - \frac{wi_{Candidate}}{2} > \frac{w_n}{2})$
 8) $Candidate = n' \mid \forall n''$ with $n', n'' \in Children, w_{n'} - \frac{wi_{n'}}{2} \geq w_{n''} - \frac{wi_{n''}}{2}$
 9) if $(M(Best) = Wrong)$ then return $Candidate$
10) if $(w_n \geq w_{Best} + w_{Candidate} - \frac{wi_{Best}}{2} - \frac{wi_{Candidate}}{2})$ then return $Best$
11)                                                 else return $Candidate$
end
```

Assigning a (total) weight of zero to a node has been used for instance in the technique called *Trusting* [65]. This technique allows the user to trust a method. When this happens all the nodes related to this method and their descendants are pruned from the tree (i.e., these nodes have a (total) weight of zero).

If we add the restriction that nodes cannot be assigned with an individual weight of zero, then we can refine Algorithm 22 to ensure completeness. This refined version is Algorithm 23.

Algorithm 23 Optimal D&Q General (Complete) —SelectNode in Algorithm 1—

```
Input: A MET $T = (N, E, M)$ whose root is $n \in N$ and $\forall n' \in N, wi_{n'} > 0$
Output: A set of nodes $O \subseteq N$
Preconditions: $\exists n' \in N, M(n') = Undefined$

begin
 1) $Candidate = n$
 2) do
 3)    $Best = Candidate$
 4)    $Children = \{m \mid (Best \to m) \in E\}$
 5)    if $(Children = \emptyset)$ then return $\{Best\}$
 6)    $Candidate = n' \mid \forall n''$ with $n', n'' \in Children, w_{n'} \geq w_{n''}$
 7) while $(w_{Candidate} - \frac{wi_{Candidate}}{2} > \frac{w_n}{2})$
 8) $Candidates = \{n' \mid \forall n''$ with $n', n'' \in Children, w_{n'} - \frac{wi_{n'}}{2} \geq w_{n''} - \frac{wi_{n''}}{2}\}$
 9) $Candidate = n' \in Candidates$
10) if $(M(Best) = Wrong)$ then return $Candidates$
11) if $(w_n > w_{Best} + w_{Candidate} - \frac{wi_{Best}}{2} - \frac{wi_{Candidate}}{2})$ then return $\{Best\}$
12) if $(w_n = w_{Best} + w_{Candidate} - \frac{wi_{Best}}{2} - \frac{wi_{Candidate}}{2})$ then return $\{Best\} \cup Candidates$
13)                                                 else return $Candidates$
end
```

6.2.6 Proofs of technical results

In this section, for the sake of clarity, we use u_n and d_n instead of $Up(n)$ and $Down(n)$ respectively. Moreover, we distinguish between two kinds of METs to prove the theorems of Sections 6.2.3 and 6.2.4, respectively.

Definition 6.2.10 (Uniform MET) *A* uniform MET $T = (N, E, M)$ *is a MET, where* $\forall n, n' \in N, wi_n = wi_{n'}$ *and* $\forall n \in N, wi_n > 0$.

Definition 6.2.11 (Variable MET) *A* variable MET $T = (N, E, M)$ *is a MET, where* $\forall n \in N, wi_n \geq 0$.

Proofs of Theorems 6.2.3, 6.2.4 and 6.2.5

Here, we prove Theorems 6.2.3, 6.2.4 and 6.2.5 that are used in Algorithm 20 to compare nodes of the MET and determine which of them is better. For the proof of Theorem 6.2.3, we need to prove first the following lemma.

Lemma 6.2.12 *Let* $T = (N, E, M)$ *be a uniform MET whose root is* $n \in N$, *and let* $n_1, n_2 \in Sea(T)$ *be two nodes. Then,* $n_1 \gg n_2$ *if and only if* $u_{n_1} * d_{n_1} > u_{n_2} * d_{n_2}$.

Proof. We prove that $u_{n_1} * d_{n_1} > u_{n_2} * d_{n_2}$ implies that $|d_{n_1} - u_{n_1}| < |d_{n_2} - u_{n_2}|$ and vice versa. This can be shown by developing the equation $u_{n_1} * d_{n_1} > u_{n_2} * d_{n_2}$.
Firstly, note that $w_n = \sum \{wi_{n'} \mid n' \in Sea(T)\}$, then by Equation 1 we know that $w_n = u_{n_1} + d_{n_1} + wi_{n_1} = u_{n_2} + d_{n_2} + wi_{n_2}$. Therefore, as $wi_{n_1} = wi_{n_2} = wi_n$ the optimal division of $Sea(T)$ happens when $u_{n_1} = d_{n_1} = \frac{w_n - wi_n}{2}$. For the sake of simplicity in the notation, let $c = \frac{w_n - wi_n}{2}$ and let $h_1 = c - d_{n_1} = u_{n_1} - c$ and $h_2 = c - d_{n_2} = u_{n_2} - c$. Then,

$u_{n_1} * d_{n_1} > u_{n_2} * d_{n_2}$
Therefore, we replace u_{n_1}, d_{n_1}, u_{n_2} *and* d_{n_2}:
$(c + h_1) * (c - h_1) > (c + h_2) * (c - h_2)$
$c^2 - h_1 * c + h_1 * c - h_1^2 > c^2 - h_2 * c + h_2 * c - h_2^2$
We simplify:
$c^2 - h_1^2 > c^2 - h_2^2$
$-h_1^2 > -h_2^2$
$h_1^2 < h_2^2$
And finally we obtain that:
$|h_1| < |h_2|$

Hence, if the product $u_{n_1} * d_{n_1}$ is greater than $u_{n_2} * d_{n_2}$ then $|h_1| < |h_2|$ and thus, because h_1 and h_2 represent distances to the centre, $n_1 \gg n_2$. □

Theorem 6.2.3. *Given a uniform MET* $T = (N, E, M)$ *whose root is* $n \in N$, *and given two nodes* $n_1, n_2 \in Sea(T)$, *with* $w_{n_1} > w_{n_2}$, $n_1 \gg n_2$ *if and only if* $w_n > w_{n_1} + w_{n_2} - wi_n$.

Proof. By Lemma 6.2.12 we know that if $u_{n_1} * d_{n_1} > u_{n_2} * d_{n_2}$ then $n_1 \gg n_2$. Thus it is enough to prove that $w_n > w_{n_1} + w_{n_2} - wi_n$ implies $u_{n_1} * d_{n_1} > u_{n_2} * d_{n_2}$ and vice versa when $w_{n_1} > w_{n_2}$.

$w_n > w_{n_1} + w_{n_2} - wi_n$
Adding $wi_n - wi_n$*:*
$w_n > w_{n_1} + w_{n_2} - 2 * wi_n + wi_n$
We replace w_{n_1}*,* w_{n_2} *by Equation 2:*
$w_n > d_{n_1} + d_{n_2} + wi_n$
Adding $wi_n * d - wi_n * d$*:*
$w_n > d_{n_1} + d_{n_2} + wi_n * d + wi_n - wi_n * d$
$w_n > d_{n_1} + d_{n_2} + wi_n * d + wi_n(1 - d)$
Using $d = \frac{d_{n_1}}{d_{n_1} - d_{n_2}}$ *we get:*
$w_n > d_{n_1} + d_{n_2} + wi_n \frac{d_{n_1}}{d_{n_1}-d_{n_2}} + wi_n(1 - \frac{d_{n_1}}{d_{n_1}-d_{n_2}})$
$w_n > d_{n_1} + d_{n_2} + wi_n \frac{d_{n_1}}{d_{n_1}-d_{n_2}} + wi_n(\frac{d_{n_1}-d_{n_2}}{d_{n_1}-d_{n_2}} - \frac{d_{n_1}}{d_{n_1}-d_{n_2}})$
$w_n > d_{n_1} + d_{n_2} + wi_n \frac{d_{n_1}}{d_{n_1}-d_{n_2}} + wi_n \frac{-d_{n_2}}{d_{n_1}-d_{n_2}}$
$w_n > d_{n_1} + d_{n_2} + wi_n \frac{d_{n_1}}{d_{n_1}-d_{n_2}} - wi_n \frac{d_{n_2}}{d_{n_1}-d_{n_2}}$
Because $d_{n_1} + d_{n_2} = \frac{d_{n_1}^2 - d_{n_2}^2}{d_{n_1}-d_{n_2}}$ *then:*
$w_n > \frac{d_{n_1}^2 - d_{n_2}^2}{d_{n_1}-d_{n_2}} + \frac{d_{n_1} * wi_n}{d_{n_1}-d_{n_2}} - \frac{d_{n_2} * wi_n}{d_{n_1}-d_{n_2}}$
Because $w_{n_1} > w_{n_2}$ *we know by Equation 2 that* $d_{n_1} - d_{n_2} > 0$*, thus:*
$(d_{n_1} - d_{n_2}) * w_n > d_{n_1}^2 - d_{n_2}^2 + d_{n_1} * wi_n - d_{n_2} * wi_n$
$d_{n_1} * w_n - d_{n_2} * w_n > d_{n_1}^2 - d_{n_2}^2 + d_{n_1} * wi_n - d_{n_2} * wi_n$
$d_{n_1} * w_n - d_{n_1}^2 - d_{n_1} * wi_n > d_{n_2} * w_n - d_{n_2}^2 - d_{n_2} * wi_n$
$d_{n_1} * (w_n - d_{n_1} - wi_n) > d_{n_2} * (w_n - d_{n_2} - wi_n)$
As $wi_n = wi_{n_1} = wi_{n_2}$ *we replace* $w_n - d_{n_1} - wi_n$*,* $w_n - d_{n_2} - wi_n$ *by Equation 1:*
$d_{n_1} * u_{n_1} > d_{n_2} * u_{n_2}$
□

Theorem 6.2.4. *Given a uniform MET* $T = (N, E, M)$ *whose root is* $n \in N$*, and given two nodes* $n_1, n_2 \in Sea(T)$*, with* $w_{n_1} > w_{n_2}$*,* $n_1 \equiv n_2$ *if and only if* $w_n = w_{n_1} + w_{n_2} - wi_n$*.*

Proof. The proof is completely analogous to the proof of Theorem 6.2.3. The only difference is that the equation that is developed should be $w_n = w_{n_1} + w_{n_2} - wi_n$. □

Theorem 6.2.5. *Let* $T = (N, E, M)$ *be a uniform MET, and let* $n_1, n_2 \in Sea(T)$ *be two nodes, if* $w_{n_1} = w_{n_2}$ *then* $n_1 \equiv n_2$*.*

Proof. We prove that $w_{n_1} = w_{n_2}$ implies $|d_{n_1} - u_{n_1}| = |d_{n_2} - u_{n_2}|$ and thus $n_1 \equiv n_2$:

$$\begin{array}{lr}
w_{n_1} = w_{n_2} & \textit{we replace } w_{n_1}, w_{n_2} \textit{ by Equation 2} \\
d_{n_1} + wi_{n_1} = d_{n_2} + wi_{n_2} & \textit{using } wi_{n_1} = wi_{n_2} \\
d_{n_1} = d_{n_2} & \textit{using } w_{n_1} = w_{n_2} \\
w_{n_1} - w_n + d_{n_1} = w_{n_2} - w_n + d_{n_2} & \textit{replacing } w_{n_1}, w_{n_2} \textit{ by Equation 2} \\
(d_{n_1} + wi_{n_1}) - (u_{n_1} + d_{n_1} + wi_{n_1}) + d_{n_1} & \textit{and } w_n \textit{ by Equation 1} \\
\qquad = (d_{n_2} + wi_{n_2}) - (u_{n_2} + d_{n_2} + wi_{n_2}) + d_{n_2} & \textit{we simplify} \\
d_{n_1} - u_{n_1} = d_{n_2} - u_{n_2} & \\
|d_{n_1} - u_{n_1}| = |d_{n_2} - u_{n_2}| &
\end{array}$$

□

Corollary 6.2.6. *Given a uniform MET* $T = (N, E, M)$*, and given a node* $n \in Sea(T)$*, then* n *optimally divides* $Sea(T)$ *if and only if* $u_n = d_n$*.*

Proof. If n optimally divides $Sea(T)$ then the product $u_n * d_n$ is maximum, and there does not exist other node $n' \in Sea(T)$ such that $u_{n'} * d_{n'} > u_n * d_n$. This can be easily shown taking into account that the figure of the product is a parabola whose vertex is the maximum value. Therefore, we can compute the maximum by deriving the product.

For simplicity, let $prod = u_n * d_n$ and $sum = u_n + d_n$. Then, we start by transforming the equation $u_n * d_n$ in such a way that it only depends on one of the factors (e.g., u_n):

$u_n * d_n = prod$
We replace d_n :
$u_n * (sum - u_n) = prod$
$u_n * sum - u_n^2 = prod$
We derive the equation and equate it to zero:
$\frac{d}{du_n}(u_n * sum - u_n^2) = 0$
$sum - 2u_n = 0$
And finally we get the value of u_n *in the vertex:*
$u_n = \frac{sum}{2}$

Now, we can infer d_n from u_n by simply replacing the value of u_n in the equation $u_n + d_n = sum$:

$\frac{sum}{2} + d_n = sum$
$d_n = sum - \frac{sum}{2}$
$d_n = \frac{sum}{2}$
$d_n = u_n$

□

Proof of Theorem 6.2.8

Theorem 6.2.8 states the correctness of Algorithm 20 used when all nodes have the same individual weight. Firstly, we prove the following auxiliary lemma.

Lemma 6.2.13 *Let $T = (N, E, M)$ be a uniform MET whose root is $n \in N$ and $n_1, n_2 \in Sea(T)$ with $w_{n_1} > w_{n_2}$, if $w_n \geq w_{n_1} + w_{n_2}$ then $n_1 \gg n_2$.*

Proof. Firstly, by Theorem 6.2.3 we know that if $w_n > w_{n_1} + w_{n_2} - wi_n$ when $w_{n_1} > w_{n_2}$ then $n_1 \gg n_2$. Therefore, as $wi_n > 0$, if $w_n \geq w_{n_1} + w_{n_2}$ then $w_n > w_{n_1} + w_{n_2} - wi_n$ and hence $n_1 \gg n_2$. □

In order to prove the correctness of Algorithm 20, we also need to prove the four cases presented in Section 6.2.3 that are used in the algorithm:

Case 1: n_1 and n_2 are siblings.
Case 2: $w_{n_1} > w_{n_2} \ \wedge \ w_{n_2} > \frac{w_n}{2}$.
Case 3: $w_{n_1} > \frac{w_n}{2} \ \wedge \ w_{n_2} \leq \frac{w_n}{2}$.
Case 4: $w_{n_1} > w_{n_2} \ \wedge \ w_{n_1} \leq \frac{w_n}{2}$.

We prove each case in a separate lemma. In case 1, the following lemma shows that given two sibling nodes n_1 and n_2, then the heaviest node is better.

Lemma 6.2.14 *Given a uniform MET $T = (N, E, M)$ whose root is $n \in N$ and given three nodes $n_1 \in N$ and $n_2, n_3 \in Sea(T)$ with $(n \rightarrow n_1) \in E^*$,$(n_1 \rightarrow n_2), (n_1 \rightarrow n_3) \in E$, $n_2 \gg n_3 \vee n_2 \equiv n_3$ if and only if $w_{n_2} \geq w_{n_3}$.*

Proof. We prove first that $w_{n_2} \geq w_{n_3}$ implies $n_2 \gg n_3 \vee n_2 \equiv n_3$: Trivially, $w_n \geq w_{n_2} + w_{n_3}$ because n_2 and n_3 are children of n_1 and n_1 is descendant of n. Therefore, by Lemma 6.2.13 and Theorem 6.2.5,

$n_2 \gg n_3 \vee n_2 \equiv n_3$. Now, we prove that $n_2 \gg n_3 \vee n_2 \equiv n_3$ implies $w_{n_2} \geq w_{n_3}$: We prove it by contradiction assuming that $w_{n_2} < w_{n_3}$ when $n_2 \gg n_3 \vee n_2 \equiv n_3$, and proving that when $w_{n_2} < w_{n_3}$ and $n_2 \gg n_3 \vee n_2 \equiv n_3$, neither $w_n > w_{n_2} + w_{n_3} - wi_n$ nor $w_n \leq w_{n_2} + w_{n_3} - wi_n$ hold. By Theorem 6.2.3 $w_n > w_{n_2} + w_{n_3} - wi_n$ is false because $n_2 \gg n_3 \vee n_2 \equiv n_3$. Moreover, because n_2 and n_3 are siblings, we know that $w_n \geq w_{n_2} + w_{n_3}$, and hence $w_n \leq w_{n_2} + w_{n_3} - wi_n$ is also false. □

In case 2, the following lemma ensures that given two nodes n_1 and n_2 such that n_1 is the parent of n_2, if $w_{n_2} > \frac{w_n}{2}$ then n_2 is better.

Lemma 6.2.15 *Given a uniform MET $T = (N, E, M)$ whose root is $n \in N$, and given two nodes $n_1, n_2 \in Sea(T)$, with $(n_1 \to n_2) \in E$, if $w_{n_2} > \frac{w_n}{2}$ then $n_2 \gg n_1$.*

Proof. We prove the lemma by contradiction assuming that $n_1 \gg n_2$ or $n_1 \equiv n_2$. First, we know that $w_{n_2} = \frac{w_n}{2} + inc_{n_2}$ with $inc_{n_2} > 0$. And we know that $w_{n_1} = \frac{w_n}{2} + inc_{n_2} + wi_n + inc_{n_1}$ with $inc_{n_1} \geq 0$, where inc_{n_1} represent the weight of the possible siblings of n_2. By Theorems 6.2.3 and 6.2.4 we know that $w_n \geq w_{n_1} + w_{n_2} - wi_n$ when $w_{n_1} > w_{n_2}$ implies $n_1 \gg n_2 \vee n_1 \equiv n_2$.

$$\begin{array}{ll}
w_n \geq w_{n_1} + w_{n_2} - wi_n & \textit{We replace } w_{n_1}, w_{n_2} \\
w_n \geq (\frac{w_n}{2} + inc_{n_2} + wi_n + inc_{n_1}) + (\frac{w_n}{2} + inc_{n_2}) - wi_n & \textit{we simplify} \\
w_n \geq \frac{w_n}{2} + inc_{n_2} + inc_{n_1} + \frac{w_n}{2} + inc_{n_2} & \\
w_n \geq \frac{w_n}{2} + \frac{w_n}{2} + 2 * inc_{n_2} + inc_{n_1} & \\
w_n \geq w_n + 2 * inc_{n_2} + inc_{n_1} & \\
0 \geq 2 * inc_{n_2} + inc_{n_1} &
\end{array}$$

But, this is a contradiction with $inc_{n_2} > 0$. Hence, $n_2 \gg n_1$. □

In case 4, the following lemma ensures that given two nodes whose weight is smaller than $\frac{w_n}{2}$ then the heaviest node is better.

Lemma 6.2.16 *Given a uniform MET $T = (N, E, M)$ whose root is $n \in N$, and two nodes $n_1, n_2 \in Sea(T)$, where $\frac{w_n}{2} \geq w_{n_1} > w_{n_2}$ then $n_1 \gg n_2$.*

Proof. We can assume that $w_{n_1} = \frac{w_n}{2} - dec_{n_1}$ and $w_{n_2} = \frac{w_n}{2} - dec_{n_2}$ where $dec_{n_2} > dec_{n_1} \geq 0$. Moreover, we know that $w_{n_1} + w_{n_2} = \frac{w_n}{2} - dec_{n_1} + \frac{w_n}{2} - dec_{n_2}$ and thus $w_{n_1} + w_{n_2} = w_n - dec_{n_1} - dec_{n_2}$. Therefore, because $dec_{n_2} > dec_{n_1} \geq 0$, we deduce that $w_n > w_{n_1} + w_{n_2}$. And as $w_{n_1} > w_{n_2}$ then, by Lemma 6.2.13, $n_1 \gg n_2$. □

If two nodes n_1 and n_2 are siblings and n_1 is better than n_2 then n_1 is better than any descendant of n_2. The following lemma proves this property that is complementary to Lemma 6.2.14 for case 1.

Lemma 6.2.17 *Given a uniform MET $T = (N, E, M)$ whose root is $n \in N$ and four nodes $n_1 \in N$ and $n_2, n_3, n_4 \in Sea(T)$ with $(n \to n_1) \in E^*$, $(n_1 \to n_2), (n_1 \to n_3) \in E$, $(n_3 \to n_4) \in E^+$, if $n_2 \gg n_3 \vee n_2 \equiv n_3$ then $n_2 \gg n_4$.*

Proof. First, n_2 and n_3 are siblings and $n_2 \gg n_3 \vee n_2 \equiv n_3$ then, by Lemma 6.2.14, we know that $w_{n_2} \geq w_{n_3}$. We distinguish two cases $w_{n_2} > \frac{w_n}{2}$ and $\frac{w_n}{2} \geq w_{n_2}$.
If $\frac{w_n}{2} \geq w_{n_2}$ then $\frac{w_n}{2} \geq w_{n_3}$ and by Lemma 6.2.16 $n_3 \gg n_4$.
If $w_{n_2} > \frac{w_n}{2}$ then we only have to demonstrate that $\frac{w_n}{2} > w_{n_3}$ and then (as before) by Lemma 6.2.16 $n_3 \gg n_4$.
This can be easily proved taking into account that $w_n \geq w_{n_2} + w_{n_3}$ because n_2 and n_3 are children of n_1 and n_1 is descendant of n, and that $w_{n_2} = \frac{w_n}{2} + inc_{n_2}$ with $inc_{n_2} > 0$.

$$\begin{array}{ll} w_n \geq w_{n_2} + w_{n_3} & \textit{we replace } w_{n_2} \\ w_n \geq (\frac{w_n}{2} + inc_{n_2}) + w_{n_3} & \\ w_n - \frac{w_n}{2} \geq inc_{n_2} + w_{n_3} & \\ \frac{w_n}{2} \geq inc_{n_2} + w_{n_3} & \textit{as } inc_{n_2} > 0 \\ \frac{w_n}{2} > w_{n_3} & \end{array}$$

Therefore as $n_2 \gg n_3 \vee n_2 \equiv n_3$ and $n_3 \gg n_4$ then $n_2 \gg n_4$. □

The previous lemmas allow Algorithm 20 to find a path between the root node and an optimal node. The correctness of this algorithm is proved by the following theorem.

Theorem 6.2.8. *Let $T = (N, E, M)$ be a uniform MET, then the execution of Algorithm 20 with T as input always terminates producing as output a node $n \in Sea(T)$ such that $\nexists n' \in Sea(T) \mid n' \gg n$.*

Proof. The finiteness of the algorithm is proved thanks to the following invariant: $w_{Candidate}$ strictly decreases in each iteration. Therefore, because N is finite, $w_{Candidate}$ will eventually become smaller or equal to $\frac{w_n}{2}$ and the loop will terminate.

The correctness can be proved showing that after any number of iterations the algorithm always finishes with an optimal node. We prove it by induction on the number of iterations performed.

(Base Case) In the base case, only one iteration is executed. If the condition in Line (5) is satisfied then the root is marked as undefined and it is trivially the optimal node. This optimal node is returned in Line (5). Otherwise, Lines (4) and (6) select the heaviest child of the root, the loop terminates and Lines (9) or (10) return the optimal node.

Note that the root node—when it is marked as *Wrong*—can only be selected in the first iteration. But even in this case, this node is never selected because the root node must have at least one child marked as *Undefined*. Thus Line (5) is not satisfied and Line (6) selects this node. If the condition of the loop is not satisfied, then Line (8) returns the child of the root.

(Induction Hypothesis) We assume as the induction hypothesis that after i iterations, the algorithm has a candidate node $Best \in Sea(T)$ such that $\forall n' \in Sea(T), (Best \rightarrow n') \notin E^*, Best \gg n'$.

(Inductive Case) We now prove that the iteration $i + 1$ of the algorithm will select a new candidate node $Candidate$ such that $Candidate \gg Best$, or it will terminate selecting an optimal node.

Firstly, when the condition in Line (5) is satisfied $Best$ and $Candidate$ are the same node (say n'). According to the induction hypothesis, this node is better than any other of the nodes in the set $\{n'' \in Sea(T) \mid (n' \rightarrow n'') \notin E^*\}$. Therefore, because n' has no children, then it is an optimal node; and it is returned in Line (5). Otherwise, if the condition in Line (5) is not satisfied, Line (7) in the algorithm ensures that $w_{Best} > \frac{w_n}{2}$ being n the root of T because in the iteration i the loop did not terminate or because $Best$ is the root. Moreover, according to Lines (4) and (6), we know that $Candidate$ is the heaviest child of $Best$. We have two possibilities:

- $w_{Candidate} > \frac{w_n}{2}$: In this case the loop does not terminate and $\forall n' \in Sea(T)$, $(Candidate \rightarrow n') \notin E^*, Candidate \gg n'$. Firstly, by Lemma 6.2.15 we know that $Candidate \gg Best$, and thus, by the induction hypothesis we know that $\forall n' \in Sea(T), (Best \rightarrow n') \notin E^*, Candidate \gg n'$. By Lemma 6.2.14 $Candidate \gg n' \vee Candidate \equiv n'$ being n' a sibling of $Candidate$. But as we know that $w_{Candidate} > \frac{w_n}{2}$ then $Candidate \not\equiv n'$. Moreover, by Lemma 6.2.17 we can ensure that $Candidate \gg n'$ being n' a descendant of a sibling of $Candidate$.

- $w_{Candidate} \leq \frac{w_n}{2}$: In this case the loop terminates (Line (7)) and by Lemma 6.2.14 we know that $Candidate \gg n' \vee Candidate \equiv n'$ being n' a sibling of $Candidate$. Moreover, by Lemma 6.2.17 we can ensure that $Candidate \gg n'$ being n' a descendant of a sibling of $Candidate$. Then equation $(w_n \geq w_{Best} + w_{Candidate} - wi_n)$ is applied in Line (9) to select an optimal node. Theorems 6.2.3 and 6.2.4 ensures that the node selected is an optimal node because, according to Lemma 6.2.16, for all descendant n' of $Candidate$, $Candidate \gg n'$.

□

Proof of Theorem 6.2.9

Theorem 6.2.9 states the correctness of Algorithm 22 used in the general case when nodes can have different individual weights. For the proof of this theorem we define first some auxiliary lemmas. The following lemma ensures that $w_{n_1} - \frac{wi_{n_1}}{2} > \frac{w_n}{2}$ used in the condition of the loop implies $d_{n_1} > u_{n_1}$.

Lemma 6.2.18 *Given a variable MET $T = (N, E, M)$ whose root is $n \in N$ and a node $n_1 \in Sea(T)$, $d_{n_1} > u_{n_1}$ if and only if $w_{n_1} - \frac{wi_{n_1}}{2} > \frac{w_n}{2}$.*

Proof. We prove that $w_{n_1} - \frac{wi_{n_1}}{2} > \frac{w_n}{2}$ implies $d_{n_1} > u_{n_1}$ and vice versa.
$w_{n_1} - \frac{wi_{n_1}}{2} > \frac{w_n}{2}$
$2w_{n_1} - wi_{n_1} > w_n$
We replace w_{n_1} using Equation 2:
$2(d_{n_1} + wi_{n_1}) - wi_{n_1} > w_n$
$2d_{n_1} + wi_{n_1} > w_n$
$d_{n_1} > w_n - d_{n_1} - wi_{n_1}$
We replace $w_n - d_{n_1} - wi_{n_1}$ using Equation 1:
$d_{n_1} > u_{n_1}$

□

The following lemma ensures that given two nodes n_1 and n_2 where $d_n \geq u_n$ in both nodes and $n_1 \rightarrow n_2$ then $n_2 \gg n_1 \vee n_2 \equiv n_1$.

Lemma 6.2.19 *Given a variable MET $T = (N, E, M)$ and given two nodes $n_1, n_2 \in Sea(T)$, with $(n_1 \rightarrow n_2) \in E$, if $d_{n_2} \geq u_{n_2}$ then $n_2 \gg n_1 \vee n_2 \equiv n_1$.*

Proof. We prove that $|d_{n_2} - u_{n_2}| \leq |d_{n_1} - u_{n_1}|$ holds. First, we know that $d_{n_1} = d_{n_2} + wi_{n_2} + inc$ and $u_{n_1} = u_{n_2} - wi_{n_1} - inc$ with $inc \geq 0$, where inc represent the weight of the possible siblings of n_2.
$|d_{n_2} - u_{n_2}| \leq |d_{n_1} - u_{n_1}|$
As we know that $d_n \geq u_n$ in both nodes:
$d_{n_2} - u_{n_2} \leq d_{n_1} - u_{n_1}$
We replace d_{n_1} and u_{n_1}:
$d_{n_2} - u_{n_2} \leq (d_{n_2} + wi_{n_2} + inc) - (u_{n_2} - wi_{n_1} - inc)$
$d_{n_2} - u_{n_2} \leq d_{n_2} - u_{n_2} + wi_{n_1} + wi_{n_2} + 2inc$
$0 \leq wi_{n_1} + wi_{n_2} + 2inc$
Hence, because wi_{n_1}, wi_{n_2}, $inc \geq 0$ then $|d_{n_2} - u_{n_2}| \leq |d_{n_1} - u_{n_1}|$ is satisfied and thus $n_2 \gg n_1 \vee n_2 \equiv n_1$.
□

The following lemma ensures that given two nodes n_1 and n_2 where $d_n \leq u_n$ in both nodes and $n_1 \rightarrow n_2$ then $n_1 \gg n_2 \vee n_1 \equiv n_2$.

Lemma 6.2.20 *Given a variable MET $T = (N, E, M)$ and given two nodes $n_1, n_2 \in Sea(T)$, with $(n_1 \rightarrow n_2) \in E$, if $d_{n_1} \leq u_{n_1}$ then $n_1 \gg n_2 \vee n_1 \equiv n_2$.*

Proof. We prove that $|d_{n_1} - u_{n_1}| \leq |d_{n_2} - u_{n_2}|$ holds. First, we know that $d_{n_2} = d_{n_1} - wi_{n_2} - inc$ and $u_{n_2} = u_{n_1} + wi_{n_1} + inc$ with $inc \geq 0$, where inc represents the weight of the possible siblings of n_2.

$|d_{n_1} - u_{n_1}| \leq |d_{n_2} - u_{n_2}|$
As we know that $u_n \geq d_n$ in both nodes:
$u_{n_1} - d_{n_1} \leq u_{n_2} - d_{n_2}$
We replace d_{n_2} and u_{n_2}:
$u_{n_1} - d_{n_1} \leq (u_{n_1} + wi_{n_1} + inc) - (d_{n_1} - wi_{n_2} - inc)$
$u_{n_1} - d_{n_1} \leq u_{n_1} - d_{n_1} + wi_{n_1} + wi_{n_2} + 2inc$
$0 \leq wi_{n_1} + wi_{n_2} + 2inc$
Hence, because wi_{n_1}, wi_{n_2}, $inc \geq 0$ then $|d_{n_1} - u_{n_1}| \leq |d_{n_2} - u_{n_2}|$ is satisfied and thus $n_1 \gg n_2 \vee n_1 \equiv n_2$.
□

The following lemma ensures that given two sibling nodes n_1 and n_2, if $d_{n_1} \geq u_{n_1}$ then $d_{n_2} \leq u_{n_2}$.

Lemma 6.2.21 *Given a variable MET $T = (N, E, M)$ whose root is $n \in N$, and given three nodes $n_1 \in N$ and $n_2, n_3 \in Sea(T)$, with $(n \rightarrow n_1) \in E^*$, $(n_1 \rightarrow n_2), (n_1 \rightarrow n_3) \in E$, if $d_{n_2} \geq u_{n_2}$ then $d_{n_3} \leq u_{n_3}$.*

Proof. We prove it by contradiction assuming that $d_{n_3} > u_{n_3}$ when $d_{n_2} \geq u_{n_2}$ and they are siblings. First, we know that as n_2 and n_3 are siblings then $u_{n_2} \geq w_{n_3}$ and $u_{n_3} \geq w_{n_2}$. Therefore, if $d_{n_3} > u_{n_3}$ then $d_{n_2} \geq u_{n_2} \geq w_{n_3} \geq d_{n_3} > u_{n_3} \geq w_{n_2} \geq d_{n_2}$ that implies $d_{n_2} > d_{n_2}$ that is a contradiction itself. □

If two nodes n_1 and n_2 are siblings and $d_{n_1} \geq u_{n_1}$ then $n_1 \gg n_2 \vee n_1 \equiv n_2$. The following lemma proves this property.

Lemma 6.2.22 *Given a variable MET $T = (N, E, M)$ whose root is $n \in N$, and given three nodes $n_1 \in N$ and $n_2, n_3 \in Sea(T)$, with $(n \rightarrow n_1) \in E^*$, $(n_1 \rightarrow n_2), (n_1 \rightarrow n_3) \in E$, if $d_{n_2} \geq u_{n_2}$ then $n_2 \gg n_3 \vee n_2 \equiv n_3$.*

Proof. We prove that $|d_{n_2} - u_{n_2}| \leq |d_{n_3} - u_{n_3}|$ holds. First, as n_2 and n_3 are siblings we know that $w_n \geq d_{n_2} + d_{n_3} + wi_{n_2} + wi_{n_3}$, then $w_n = d_{n_2} + d_{n_3} + wi_{n_2} + wi_{n_3} + inc$ with $inc \geq 0$.
$|d_{n_2} - u_{n_2}| \leq |d_{n_3} - u_{n_3}|$
As $d_{n_2} \geq u_{n_2}$ by Lemma 6.2.21 we know that $u_{n_3} \geq d_{n_3}$:
$d_{n_2} - u_{n_2} \leq u_{n_3} - d_{n_3}$
We replace u_{n_2} and u_{n_3} using Equation 1:
$d_{n_2} - (w_n - d_{n_2} - wi_{n_2}) \leq (w_n - d_{n_3} - wi_{n_3}) - d_{n_3}$
$-w_n + 2d_{n_2} + wi_{n_2} \leq w_n - 2d_{n_3} - wi_{n_3}$
$-2w_n \leq -2d_{n_2} - 2d_{n_3} - wi_{n_2} - wi_{n_3}$
$2w_n \geq 2d_{n_2} + 2d_{n_3} + wi_{n_2} + wi_{n_3}$
$w_n \geq d_{n_2} + d_{n_3} + \frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2}$
We replace w_n:
$d_{n_2} + d_{n_3} + wi_{n_2} + wi_{n_3} + inc \geq d_{n_2} + d_{n_3} + \frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2}$
$wi_{n_2} + wi_{n_3} + inc \geq \frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2}$
$\frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2} + inc \geq 0$
Hence, because wi_{n_2}, wi_{n_3}, $inc \geq 0$ then $|d_{n_2} - u_{n_2}| \leq |d_{n_3} - u_{n_3}|$ is satisfied and thus $n_2 \gg n_3 \vee n_2 \equiv n_3$.
□

The following lemma ensures that given two sibling nodes n_1 and n_2, if $w_{n_1} \geq w_{n_2}$ and $d_{n_1} \leq u_{n_1}$ then $d_{n_2} \leq u_{n_2}$.

Lemma 6.2.23 *Given a variable MET $T = (N, E, M)$ whose root is $n \in N$, and given three nodes $n_1 \in N$ and $n_2, n_3 \in Sea(T)$, with $(n \rightarrow n_1) \in E^*$, $(n_1 \rightarrow n_2), (n_1 \rightarrow n_3) \in E$, if $w_{n_2} \geq w_{n_3}$ and $d_{n_2} \leq u_{n_2}$ then $d_{n_3} \leq u_{n_3}$.*

Proof. We prove it by contradiction assuming that $d_{n_3} > u_{n_3}$ when $w_{n_2} \geq w_{n_3}$ and $d_{n_2} \leq u_{n_2}$ and they are siblings. First, we know that as n_2 and n_3 are siblings then $u_{n_2} \geq w_{n_3}$ and $u_{n_3} \geq w_{n_2}$. Therefore, if $d_{n_3} > u_{n_3}$ then $d_{n_3} > u_{n_3} \geq w_{n_2} \geq w_{n_3} \geq d_{n_3}$ that implies $d_{n_3} > d_{n_3}$ that is a contradiction itself. □

If two nodes n_1 and n_2 are siblings and $u_{n_1} \geq d_{n_1} \wedge u_{n_2} \geq d_{n_2}$ then, if $w_{n_1} - \frac{wi_{n_1}}{2} \geq w_{n_2} - \frac{wi_{n_2}}{2}$ is satisfied then $n_1 \gg n_2 \vee n_1 \equiv n_2$. The following lemma proves this property.

Lemma 6.2.24 *Given a variable MET $T = (N, E, M)$ whose root is $n \in N$, and given three nodes $n_1 \in N$ and $n_2, n_3 \in Sea(T)$, with $(n \to n_1) \in E^*$, $(n_1 \to n_2), (n_1 \to n_3) \in E$, and $u_{n_2} \geq d_{n_2}$ and $u_{n_3} \geq d_{n_3}$, $n_2 \gg n_3 \vee n_2 \equiv n_3$ if and only if $w_{n_2} - \frac{wi_{n_2}}{2} \geq w_{n_3} - \frac{wi_{n_3}}{2}$.*

Proof. First, if $|d_{n_2} - u_{n_2}| \leq |d_{n_3} - u_{n_3}|$ then $n_2 \gg n_3 \vee n_2 \equiv n_3$. Thus it is enough to prove that $w_{n_2} - \frac{wi_{n_2}}{2} \geq w_{n_3} - \frac{wi_{n_3}}{2}$ implies $|d_{n_2} - u_{n_2}| \leq |d_{n_3} - u_{n_3}|$ and vice versa when $u_n \geq d_n$ in both nodes and they are siblings.

$w_{n_2} - \frac{wi_{n_2}}{2} \geq w_{n_3} - \frac{wi_{n_3}}{2}$
$2w_{n_2} - wi_{n_2} \geq 2w_{n_3} - wi_{n_3}$
We replace w_{n_2} and w_{n_3} using Equation 2:
$2(d_{n_2} + wi_{n_2}) - wi_{n_2} \geq 2(d_{n_3} + wi_{n_3}) - wi_{n_3}$
$2d_{n_2} + wi_{n_2} \geq 2d_{n_3} + wi_{n_3}$
We add $-w_n$:
$-w_n + 2d_{n_2} + wi_{n_2} \geq -w_n + 2d_{n_3} + wi_{n_3}$
$w_n - 2d_{n_2} - wi_{n_2} \leq w_n - 2d_{n_3} - wi_{n_3}$
We replace w_n using Equation 1:
$(d_{n_2} + u_{n_2} + wi_{n_2}) - 2d_{n_2} - wi_{n_2} \leq (d_{n_3} + u_{n_3} + wi_{n_3}) - 2d_{n_3} - wi_{n_3}$
$-d_{n_2} + u_{n_2} \leq -d_{n_3} + u_{n_3}$
$u_{n_2} - d_{n_2} \leq u_{n_3} - d_{n_3}$
As $u_n \geq d_n$ in both nodes:
$|u_{n_2} - d_{n_2}| \leq |u_{n_3} - d_{n_3}|$
$|d_{n_2} - u_{n_2}| \leq |d_{n_3} - u_{n_3}|$
□

If two nodes n_1 and n_2 are siblings and $d_{n_1} \geq u_{n_1}$ and n_2 is an ancestor of n_3 then, if $n_1 \equiv n_2$ then $n_1 \gg n_3 \vee n_1 \equiv n_3$. The following lemma proves this property.

Lemma 6.2.25 *Given a variable MET $T = (N, E, M)$ whose root is $n \in N$, and given four nodes $n_1 \in N$ and $n_2, n_3, n_4 \in Sea(T)$, with $(n \to n_1) \in E^*$, $(n_1 \to n_2), (n_1 \to n_3) \in E$, $(n_3 \to n_4) \in E^+$, if $d_{n_2} \geq u_{n_2}$ and $n_2 \equiv n_3$ then $n_2 \gg n_4 \vee n_2 \equiv n_4$.*

Proof. This can be trivially proved taking into account that $d_{n_3} \leq u_{n_3}$ when $d_{n_2} \geq u_{n_2}$ by Lemma 6.2.21 and then by Lemma 6.2.20 we know that $n_3 \gg n_4 \vee n_3 \equiv n_4$ and as $n_2 \equiv n_3$ then $n_2 \gg n_4 \vee n_2 \equiv n_4$. □

If two nodes n_1 and n_2 are siblings and $d_{n_1} \leq u_{n_1} \wedge d_{n_2} \leq u_{n_2}$ and n_2 is an ancestor of n_3 then, if $n_1 \equiv n_2$ then $n_1 \gg n_3 \vee n_1 \equiv n_3$. The following lemma proves this property.

Lemma 6.2.26 *Given a variable MET $T = (N, E, M)$ whose root is $n \in N$, and given four nodes $n_1 \in N$ and $n_2, n_3, n_4 \in Sea(T)$, with $(n \to n_1) \in E^*$, $(n_1 \to n_2), (n_1 \to n_3) \in E$, $(n_3 \to n_4) \in E^+$, if $d_{n_2} \leq u_{n_2}$ and $d_{n_3} \leq u_{n_3}$ and $n_2 \equiv n_3$ then $n_2 \gg n_4 \vee n_2 \equiv n_4$.*

Proof. This can be trivially proved taking into account that $d_{n_3} \leq u_{n_3}$ and then by Lemma 6.2.20 we know that $n_3 \gg n_4 \vee n_3 \equiv n_4$ and as $n_2 \equiv n_3$ then $n_2 \gg n_4 \vee n_2 \equiv n_4$. □

If two nodes n_1 and n_2 are siblings and $n_1 \gg n_2$ and n_2 is an ancestor of n_3 then $n_1 \gg n_3$. The following lemma proves this property.

Lemma 6.2.27 *Given a variable MET $T = (N, E, M)$ whose root is $n \in N$, and given four nodes $n_1 \in N$ and $n_2, n_3, n_4 \in Sea(T)$, with $(n \rightarrow n_1) \in E^*$, $(n_1 \rightarrow n_2), (n_1 \rightarrow n_3) \in E$, $(n_3 \rightarrow n_4) \in E^+$, if $n_2 \gg n_3$ then $n_2 \gg n_4$.*

Proof. We show that if $n_2 \gg n_3$ then $d_{n_3} < u_{n_3}$. We prove it by contradiction assuming that $d_{n_3} \geq u_{n_3}$ when $n_2 \gg n_3$. First, as n_2 and n_3 are siblings we know that $w_n \geq d_{n_2} + d_{n_3} + wi_{n_2} + wi_{n_3}$, then $w_n = d_{n_2} + d_{n_3} + wi_{n_2} + wi_{n_3} + inc$ with $inc \geq 0$. Therefore, if $|d_{n_2} - u_{n_2}| < |d_{n_3} - u_{n_3}|$ then $n_2 \gg n_3$. Thus it is enough to prove that $|d_{n_2} - u_{n_2}| < |d_{n_3} - u_{n_3}|$ is not satisfied when $d_{n_3} \geq u_{n_3}$ and n_2 and n_3 are siblings.

$|d_{n_2} - u_{n_2}| < |d_{n_3} - u_{n_3}|$
As $d_{n_3} \geq u_{n_3}$ by Lemma 6.2.21 we know that $u_{n_2} \geq d_{n_2}$:
$u_{n_2} - d_{n_2} < d_{n_3} - u_{n_3}$
We replace u_{n_2} and u_{n_3} using Equation 1:
$(w_n - d_{n_2} - wi_{n_2}) - d_{n_2} < d_{n_3} - (w_n - d_{n_3} - wi_{n_3})$
$w_n - 2d_{n_2} - wi_{n_2} < 2d_{n_3} - w_n + wi_{n_3}$
$2w_n < 2d_{n_2} + 2d_{n_3} + wi_{n_2} + wi_{n_3}$
$w_n < d_{n_2} + d_{n_3} + \frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2}$
We replace w_n:
$d_{n_2} + d_{n_3} + wi_{n_2} + wi_{n_3} + inc < d_{n_2} + d_{n_3} + \frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2}$
$wi_{n_2} + wi_{n_3} + inc < \frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2}$
$\frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2} + inc < 0$
But, this is a contradiction with $wi_{n_2}, wi_{n_3}, inc \geq 0$. Hence, $d_{n_3} < u_{n_3}$.

Now we show that, if $n_2 \gg n_3$ then $n_2 \gg n_4$. We prove it by contradiction assuming that $n_4 \gg n_2 \vee n_4 \equiv n_2$ when $n_2 \gg n_3$. First, we know that $d_{n_3} < u_{n_3}$. Therefore we know that $d_{n_4} = d_{n_3} - wi_{n_4} - dec$ and $u_{n_4} = u_{n_3} + wi_{n_3} + dec$ with $dec \geq 0$, where dec represent the weight of the possible siblings of n_4.

$|d_{n_3} - u_{n_3}| > |d_{n_2} - u_{n_2}| \geq |d_{n_4} - u_{n_4}|$
We replace d_{n_4} and u_{n_4}:
$|d_{n_3} - u_{n_3}| > |d_{n_2} - u_{n_2}| \geq |(d_{n_3} - wi_{n_4} - dec) - (u_{n_3} + wi_{n_3} + dec)|$
$|d_{n_3} - u_{n_3}| > |d_{n_2} - u_{n_2}| \geq |d_{n_3} - wi_{n_4} - dec - u_{n_3} - wi_{n_3} - dec|$
$|d_{n_3} - u_{n_3}| > |d_{n_2} - u_{n_2}| \geq |d_{n_3} - u_{n_3} - wi_{n_3} - wi_{n_4} - 2dec|$
Note that $d_{n_3} - u_{n_3}$ must be positive, thus $d_{n_3} > u_{n_3}$. But this is a contradiction with $d_{n_3} < u_{n_3}$. □

The following lemma ensures that given two nodes n_1 and n_2 where $d_{n_1} \geq u_{n_1}$ and $d_{n_2} \leq u_{n_2}$ and n_1 is the parent of n_2 then if $w_n \geq w_{n_1} + w_{n_2} - \frac{wi_{n_1}}{2} - \frac{wi_{n_2}}{2}$ is satisfied then $n_1 \gg n_2 \vee n_1 \equiv n_2$.

Lemma 6.2.28 *Given a variable MET $T = (N, E, M)$ and given two nodes $n_1, n_2 \in Sea(T)$, with $(n_1 \rightarrow n_2) \in E$, and $d_{n_1} \geq u_{n_1}$, and $d_{n_2} \leq u_{n_2}$, $n_1 \gg n_2 \vee n_1 \equiv n_2$ if and only if $w_n \geq w_{n_1} + w_{n_2} - \frac{wi_{n_1}}{2} - \frac{wi_{n_2}}{2}$.*

Proof. First, if $|d_{n_1} - u_{n_1}| \leq |d_{n_2} - u_{n_2}|$ then $n_1 \gg n_2$ or $n_1 \equiv n_2$. Thus it is enough to prove that $w_n \geq w_{n_1} + w_{n_2} - \frac{wi_{n_1}}{2} - \frac{wi_{n_2}}{2}$ implies $|d_{n_1} - u_{n_1}| \leq |d_{n_2} - u_{n_2}|$ and vice versa when $d_{n_1} \geq u_{n_1}$ and $d_{n_2} \leq u_{n_2}$.

$w_n \geq w_{n_1} + w_{n_2} - \frac{wi_{n_1}}{2} - \frac{wi_{n_2}}{2}$
We replace w_{n_1}, w_{n_2} using Equation 2:
$w_n \geq (d_{n_1} + wi_{n_1}) + (d_{n_2} + wi_{n_2}) - \frac{wi_{n_1}}{2} - \frac{wi_{n_2}}{2}$
$w_n \geq d_{n_1} + d_{n_2} + \frac{wi_{n_1}}{2} + \frac{wi_{n_2}}{2}$
$2w_n \geq 2d_{n_1} + 2d_{n_2} + wi_{n_1} + wi_{n_2}$
$-2w_n \leq -2d_{n_1} - 2d_{n_2} - wi_{n_1} - wi_{n_2}$
$-w_n + 2d_{n_1} + wi_{n_1} \leq w_n - 2d_{n_2} - wi_{n_2}$
We replace w_n using Equation 1:
$-(d_{n_1} + u_{n_1} + wi_{n_1}) + 2d_{n_1} + wi_{n_1} \leq (d_{n_2} + u_{n_2} + wi_{n_2}) - 2d_{n_2} - wi_{n_2}$
$-d_{n_1} - u_{n_1} - wi_{n_1} + 2d_{n_1} + wi_{n_1} \leq d_{n_2} + u_{n_2} + wi_{n_2} - 2d_{n_2} - wi_{n_2}$
$-u_{n_1} + d_{n_1} \leq -d_{n_2} + u_{n_2}$
$d_{n_1} - u_{n_1} \leq u_{n_2} - d_{n_2}$
As $d_{n_1} \geq u_{n_1}$ and $d_{n_2} \leq u_{n_2}$:
$|d_{n_1} - u_{n_1}| \leq |u_{n_2} - d_{n_2}|$
$|d_{n_1} - u_{n_1}| \leq |d_{n_2} - u_{n_2}|$
□

Finally, we prove the correctness of Algorithm 22.

Theorem 6.2.9. *Let $T = (N, E, M)$ be a variable MET, then the execution of Algorithm 22 with T as input always terminates producing as output a node $n \in Sea(T)$ such that $\nexists n' \in Sea(T) \mid n' \gg n$.*

Proof. The finiteness of the algorithm is proved thanks to the following invariant: each iteration processes one single node, and the same node is never processed again. Therefore, because N is finite, the loop will terminate.

The proof of correctness is completely analogous to the proof of Theorem 6.2.8. The only difference is the induction hypothesis and the inductive case:

(Induction Hypothesis) After i iterations, the algorithm has a candidate node $Best \in Sea(T)$ such that $\forall n' \in Sea(T), (Best \rightarrow n') \notin E^*, Best \gg n' \vee Best \equiv n'$.

(Inductive Case) We prove that the iteration $i+1$ of the algorithm will select a new candidate node $Candidate$ such that $Candidate \gg Best \vee Candidate \equiv Best$, or it will terminate selecting an optimal node.
Firstly, when the condition in Line (5) is satisfied $Best$ and $Candidate$ are the same node (say n'). According to the induction hypothesis, this node is better or equal than any other of the nodes in the set $\{n'' \in Sea(T) | (n' \rightarrow n'') \notin E^*\}$. Therefore, because n' has no children, then it is an optimal node; and it is returned in Line (5). Otherwise, if the condition in Line (5) is not satisfied, Line (7) in the algorithm ensures that $w_{Best} - \frac{wi_{Best}}{2} > \frac{w_n}{2}$ being n the root of T because in the iteration i the loop did not terminate or because $Best$ is the root (observe that an exception can happen when all nodes have an individual weight of 0. But in this case all nodes are optimal, and thus the node returned by the algorithm is optimal). Then we know that $d_{Best} > u_{Best}$ by Lemma 6.2.18. Moreover, according to Lines (4) and (6), we know that $Candidate$ is the heaviest child of $Best$. We have two possibilities:

- $d_{Candidate} > u_{Candidate}$: In this case the loop does not terminate and $\forall n' \in Sea(T)$, $(Candidate \rightarrow n') \notin E^*, Candidate \gg n' \vee Candidate \equiv n'$. Firstly, by Lemma 6.2.19 we know that $Candidate \gg Best \vee Candidate \equiv Best$, and thus, by the induction hypothesis we know that $\forall n' \in Sea(T), (Best \rightarrow n') \notin E^*, Candidate \gg n' \vee Candidate \equiv n'$. By Lemma 6.2.22 we know that $Candidate \gg n' \vee Candidate \equiv n'$ being n' a sibling of $Candidate$. Moreover, by Lemma 6.2.25 and 6.2.27 we can ensure that $Candidate \gg n' \vee Candidate \equiv n'$ being n' a descendant of a sibling of $Candidate$.

- $d_{Candidate} \leq u_{Candidate}$: In this case the loop terminates (Line (7)) and we know by Lemma 6.2.23 that $d_{n'} \leq u_{n'}$ being n' any sibling of *Candidate*. In Line (8) according to Lemma 6.2.24 we select the Candidate such that $Candidate \gg n' \vee Candidate \equiv n'$ being n' a sibling of *Candidate*. Moreover, by Lemma 6.2.26 and 6.2.27 we can ensure that $Candidate \gg n' \vee Candidate \equiv n'$ being n' a descendant of a sibling of *Candidate*. Then equation ($w_n \geq w_{Best} + w_{Candidate} - \frac{wi_{Best}}{2} - \frac{wi_{Candidate}}{2}$) is applied in Line (10) to select an optimal node. Lemma 6.2.28 ensures that the selected node is an optimal node because, according to Lemma 6.2.20, for all descendant n' of $Candidate$, $Candidate \gg n' \vee Candidate \equiv n'$.

□

6.3 Implementation of Optimal Divide & Query

6.3.1 Introduction

The internal algorithm used in Algorithmic Debugging (AD) to decide which nodes of the Marked Execution Tree (MET) should be asked about is crucial for the performance of the technique. In [48, 50] (see Section 6.2), we conducted a series of experiments to compare the performance of different algorithms, and *Divide & Query* (D&Q) and its variants [95] showed the best performance. In that work, it is also proved that the *Optimal D&Q* variant asks (as an average) less questions.

In this section we present an implementation of Optimal D&Q [51] that has been integrated into the Declarative Debugger for Java [46]. We show how to implement this algorithm in different implementation contexts that are present in different architectures.

The rest of the section has been organized as follows. In Section 6.3.2 we recall our improved version of D&Q called Optimal D&Q. Then, in Section 6.3.3 we discuss the implementation of Optimal D&Q. Finally, Sections 6.3.4, 6.3.5 and 6.3.6 show the specific changes needed to adapt the implementation to standard, static MET and dynamic MET architectures, respectively.

6.3.2 Optimal Divide & Query

In [48, 50], Optimal Divide & Query was introduced as a new variant of D&Q that optimally divides the remaining tree with every answer. It is presented in Algorithm 24 where w_n represents the weight of node n (i.e., the weight of the subtree rooted at n), and wi_n represents the individual weight of node n (i.e., the weight of the single node n without taking into account its descendants). It is important to note that, in this algorithm, the weight of a subtree with root n is computed with the sum of the individual weights of all nodes in the subtree, but the individual weight of n is only added if it is marked as Undefined. In the case that it is marked as Wrong, then it is ignored.

Algorithm 24 Optimal D&Q (SelectNode)

```
Input: A MET $T = (N, E, M)$ whose root is $n \in N$,
       $\forall n_1, n_2 \in N, wi_{n_1} = wi_{n_2}$ and $\forall n_1 \in N, wi_{n_1} > 0$
Output: A node $n' \in N$

begin
  1) $Candidate = n$
  2) do
  3)    $Best = Candidate$
  4)    $Children = \{m \mid (Best \to m) \in E\}$
  5)    if ($Children = \emptyset$) then return $Best$
  6)    $Candidate = n' \in Children \mid \forall n'' \in Children, w_{n'} \geq w_{n''}$
  7) while ($w_{Candidate} > \frac{w_n}{2}$)
  8) if ($M(Best) = Wrong$) then return $Candidate$
  9) if ($w_n \geq w_{Best} + w_{Candidate} - wi_n$)
 10) then return $Best$
 11)  else return $Candidate$
end
```

Essentially, Algorithm 24 traverses the MET top-down from the root until it finds the buggy node. In order to do this, it compares nodes to discard some of them and define a path until the buggy node. It is based on four properties that are summarized in Figure 6.11: In cases 1 and 4, the heaviest node is better. In case 2, the lightest node is better. And in case 3, the best node must be determined with the equation $w_{root} \geq w_{n_1} + w_{n_2} - wi_{root}$ that is implemented in Line (9) of the algorithm. Observe that these cases allow the algorithm to determine the path to the optimal node that is closer to the root by comparing a reduced number of nodes.

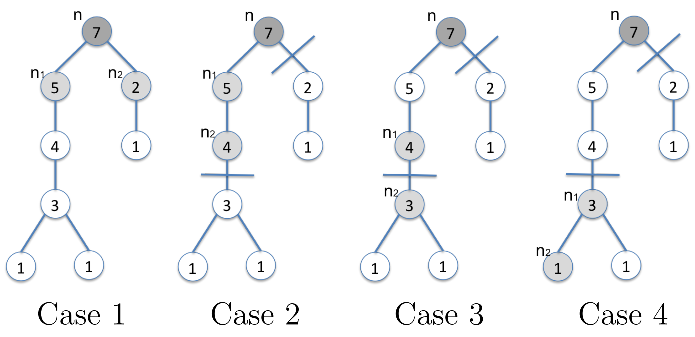


Figure 6.11: Path to the best node in a MET

6.3.3 Implementation of Optimal Divide & Query

In this section we present our implementation of the Optimal D&Q strategy and in the following sections we discuss how it can be adapted to different architectural contexts. Our algorithmic debugger, the Declarative Debugger for Java (DDJ) [46], debugs Java programs. Consequently, in this debugger the information of the nodes contain method executions and their effects.

Along this section, we assume the existence of an object that implements the Optimal D&Q strategy. We will refer to this object with the usual reference *this*, and thus, we can access the methods of this object as usual (e.g., $this.node.getState()$, $this.moveNodeToChild(indexChild)$, etc.).

The presented implementation includes the methods that compose the *OptimalDivideQuery* class. Some of these methods must be adapted depending on the architecture of the host debugger. In particular, we discuss these methods for three different architectures:

1. *Standard architecture.* Where nodes are pruned after every answer according to Algorithm 1; and where all the nodes of the MET are already generated when the first question is asked.

2. *Static MET architecture.* Where the MET is never pruned; and thus nodes have a state attribute that indicates to the search strategy whether this node can be buggy or not. Here again, it is assumed that all the nodes of the MET are already generated when the first question is asked.

3. *Dynamic MET architecture.* Where nodes are pruned after every answer according to Algorithm 1; and where the nodes in the MET can be dynamically generated while the questions are asked.

The code of the *OptimalDivideQuery* class, which is common for the three architectures, is shown in Frame 1. The code that is only used in the standard architecture is shown in Frame 2 (it continues the code of Frame 1). The code that is only used in the static MET architecture is shown in Frame 3 (it continues the code of Frame 1). The code that is only used in the dynamic MET architecture is shown in Frame 4 (it continues the code of Frame 1).

Clearly, the search strategy needs a mechanism to explore the MET and extract the information from nodes. For this, we use a pointer that can point to any node in the MET. This pointer is the *node* attribute of the search strategy that initially points to the root of the MET and can be moved to any node by means of the *selectNode* method. This method (*selectNode*) implements the Optimal D&Q strategy presented in Algorithm 24, and it uses 7 different methods during its execution:

- **node.getState()** returns the state of the current node:

```
private Node node;

public Node selectNode()
begin
this.node = this.root;
double weight = this.calculateWeight();
double weightParent = weight;
boolean thru = true;

mainLoop:
do {
    int numChildren = this.getNodeChildren();

    // Heaviest child
    int indexChild = -1;
    double weightChild = -1;
    for (int index = 0; index < numChildren; index++) {
        this.moveNodeToChild(index);
        double weightCandidate = this.node.getWeight();
        if (this.node.getState() == Undefined && weightCandidate > weightChild)
        {
            weightChild = weightCandidate;
            indexChild = index;
        }
        this.moveNodeToParent();
    }

    // Leaf
    if (indexChild == -1)
        return this.node;

    // Continue going down or equation
    thru = weightChild > weight / 2;
    if (thru || this.node.getState() == Wrong ||
    !this.equation(weight, weightParent, weightChild, this.root.getIndividualWeight()))
    {
        this.moveNodeToChild(indexChild);
        weightParent = weightChild;
    }
} while (thru);

return this.node;
end

private boolean equation(double rootWeight, double weightNode1, double weightNode2, double individualWeight)
begin
if (weightNode1 == weightNode2)
    return true;
if (weightNode1 > weightNode2)
    return rootWeight ≥ weightNode1 + weightNode2 - individualWeight;
return rootWeight < weightNode1 + weightNode2 - individualWeight;
end
```

Frame 1: *OptimalDivideQuery* class (code independent of the architecture)

- **Undefined** if no information is known about this node,
- **Wrong** if this node has been marked as wrong,
- **Right** if this node has been marked as correct,

  - **Trusted** if the method associated with this node is trusted [65] and, therefore, cannot contain the bug, and
  - **Unknown** if this node has been marked as unknown, e.g., because its associated question was too difficult.

- **node.getIndividualWeight()** returns the probability that the node contains a bug,
- **node.getWeight()** returns the probability that the subtree of the node contains a bug,
- **this.getNodeChildren()** returns the number of children from the node attribute,
- **this.moveNodeToChild(index)** updates the node attribute in order to point to one of its children,
- **this.moveNodeToParent()** updates the node attribute in order to point to its parent,
- **this.calculateWeight()** calculates the weight of the node attribute taking its state into account.

Only four of these methods depend on the selected architecture from the three discussed above. Therefore, these methods are implemented at the level of the search strategy (i.e., *this.getNodeChildren()*, *this.moveNodeToChild(index)*, *this.moveNodeToParent()* and *this.calculateWeight()*) and not at the level of a node (i.e., *node.getState()*, *node.getWeight()* and *node.getIndividualWeight()*).

The code of *selectNode()* is divided into three parts: Lines (1) to (4) initialize the main loop. Here, the *weight* variable represents the weight of the root, the *weightParent* variable represents the weight of the best node located so far and it is initialized with the weight of the root node, and the *thru* variable is used to decide when the loop must terminate; Lines (6) to (34) implement a loop that finds the optimal node; and finally Line (36) returns the optimal node.

The loop traverses the MET in a top-down manner in order to find the optimal node. We start the search in the root node and in each iteration we descend to one of its children. Once we have selected one child, the other children are discarded. Lines (8) to (22) determine this child and stores it in variable *indexChild*. If this child does not exist (e.g., the node is a leaf) then Lines (24) to (26) return the node itself as the optimal node.

After selecting the child, Line (29) checks whether the optimal node is in the subtree of the child (otherwise it must be in the child itself or in its parent). If the optimal node is in the subtree then the *thru* variable remains true, and another iteration is performed to look for the buggy node in this subtree. Otherwise, the *thru* variable becomes false and the *equation* method is used to determine the optimal node. If it returns true, the *node* attribute remains pointing to the parent, otherwise it is updated to point to the child. Finally, the node pointed by the *node* attribute is returned as the optimal node.

Note that the behaviour of the search strategy should be the same when the *thru* variable remains true and when the *equation* method returns false. In both cases the *node* attribute should be updated to point to the child. Algorithm 24 directly finishes the loop at this point and then it checks which node is the optimal one. But in our code this procedure would repeat Lines (30) and (31) after the loop. This can be easily avoided extending the *if* condition, because during the execution of Line (31) the value of *thru* is not modified and then the loop will terminate.

Another important part of the code is implemented by the *equation* method that implements Line (9) of Algorithm 24. It is used to decide which node is better between any pair of nodes. If it returns true the first node is better, otherwise the second node is better.

The following sections discuss specific changes in the algorithms that are used for each of the discussed architectures.

6.3.4 Standard architecture

The specific code for this architecture is shown in Frame 2. In this setting the nodes that cannot be buggy are pruned from the MET. This means that the root node can be marked as *Undefined* or *Wrong*, and the others can only be marked as *Undefined*. This property strongly simplifies the technique and the implementation of the methods.

```
private int getNodeChildren()
begin
(1)  return this.node.children.size();
end

private void moveNodeToChild(int index)
begin
(1)  this.node = this.node.getChild(index);
end

private void moveNodeToParent()
begin
(1)  this.node = this.node.getParent();
end

private double calculateWeight()
begin
(1)  double weight = this.node.getWeight();
(2)  if (this.node.getState() == Wrong)
(3)      weight -= this.node.getIndividualWeight();
(4)  return weight;
end
```

Frame 2: OptimalDivideQuery class (code for the standard architecture)

Traditionally, the weight of a node represents the number of nodes of the subtree. In our approach the weight of a node represents the probability that the subtree of this node contains the bug. This probability is computed with the *calculateWeight* method shown in Frame 2. This method does not take into account the wrong root when computing probabilities because it is not necessary to answer it again to determine whether it is wrong.

In this architecture, Lines (24) to (26) of Frame 1 could be moved to Line (9) using the condition *numChildren == 0* instead of *indexChild == -1* and the condition *this.node.getState() == Undefined* in line (16) can be removed. These changes make the algorithm to avoid unnecessary checks.

6.3.5 Static MET architecture

As stated before, in the standard architecture nodes are pruned from the MET. While this simplifies the technique, it also removes the possibility of reusing the same MET in other sessions (e.g., in order to find more than one bug). For this reason many debuggers avoid the pruning of the MET with some labelling mechanism that labels nodes with “possibly-buggy” or “no-buggy”. In order to make this labelling mechanism compatible with the code in Frame 1, we assign a *weight* of 0 to Right and Trusted nodes, Unknown nodes are assigned an *individual weight* of 0, and Undefined and Wrong nodes maintain their usual weights.

In an architecture where no nodes are pruned and different debugging sessions can use the same MET, a debugging session can end up with multiple Wrong nodes. This means that, in order to maintain the standard behaviour, the last node marked as wrong should be the initial (root) node of the search

```
private Node parent = null;
private List<Node> children = new ArrayList<Node>();

public Node selectNode()
begin
(1)   ...
(15)        if (this.node.getState() == Wrong) {
(16)            weight = this.calculateWeight();
(17)            weightParent = weight;
(18)            continue mainLoop;
(19)        }
(20)  ...
end

private int getNodeChildren()
begin
(1)  this.children.clear();
(2)  this.parent = this.node;
(3)  int numChildren = this.getNodeChildren(0);
(4)  this.node = this.parent;
(5)  return numChildren;
end

private int getNodeChildren(int numChildren)
begin
(1)  for (Node child : this.node.children)
(2)      if (child.getState() == Undefined || child.getState() == Wrong)
(3)          this.children.add(numChildren++, child);
(4)      else if (child.getState() == Unknown) {
(5)          this.node = child;
(6)          numChildren = this.getNodeChildren(numChildren);
(7)      }
(8)  return numChildren;
end

private void moveNodeToChild(int index)
begin
(1)  this.node = this.children.get(index);
end

private void moveNodeToParent()
begin
(1)  this.node = this.parent;
end

private double calculateWeight()
begin
(1)  double weight = this.node.getWeight();
(2)  if (this.node.getState() == Wrong)
(3)      weight -= this.node.getIndividualWeight();
(4)  return weight;
end
```

Frame 3: OptimalDivideQuery class (code for the static MET architecture)

strategy (Line (1) of *selectNode* method in Frame 1). However, some debuggers [85, 10, 23] allow the user to make manual debugging sessions. In these cases the user can select a node whose subtree already has a wrong node. The algorithm of the standard architecture is not prepared for METs with multiple wrong nodes, but this situation can be easily handled adding Lines (15) to (19) in Frame 3 between Lines (14) and (15) of the *selectNode* method in Frame 1. These lines make the algorithm to restart the search in the subtree whose root is already marked as wrong. In order to do this, we use the wrong node as the new root of the algorithm (this is done implicitly), we update both *weight* and *weightParent* to the weight of the new root, and we re-execute the external loop.

In addition, when the user marks a node as unknown, this node should be removed from the MET. But, because we do not prune nodes, they remain in the middle of the tree. This means that the *getNodeChildren* method should be modified in order to exclude Unknown nodes. Moreover, it should also be modified to exclude Right and Trusted nodes that also remain in the MET. This is performed by the new version of *getNodeChildren* shown in Frame 3, that stores in the *children* attribute the children of the current node. If one of its children is an Unknown node, it is ignored, but the children of this Unknown node are also added. After collecting the children of the current node, we can use *moveNodeToChild* and *moveNodeToParent* methods in Frame 3 to move from parent to children and vice versa. Finally, having correctly updated the weights of Right, Trusted and Unknown nodes, the *calculateWeight* method can be implemented as the one in Frame 2.

6.3.6 Dynamic MET architecture

In this section we show the changes that we should make in order to adapt the algorithm to debuggers where the MET is dynamically generated while the debugging session is performed. These debuggers allow the user to start the debugging session while the MET is being produced and thus it is uncompleted. Therefore, the MET contains:

1. **Completed nodes**. Those nodes that have been invoked and their execution already finished,
2. **Not completed nodes**. Those nodes that have been invoked but their execution did not finish, and
3. **Not generated nodes**. Those nodes that are not present in the MET because their invocation has not been performed yet.

An algorithmic debugger can only ask questions about completed nodes, that contain not only arguments and initial context but also final context and return value. Therefore, the *getNodeChildren(int numChildren)* method should exclude those nodes that are not completed yet. The implementation of this method is similar to the one from Frame 3 changing Lines (2) and (4) (see Frame 4). Note that both approaches can be used together by using a && operator in Line (2) and changing Line (4) by **else if** *(!node.isCompleted() || node.getState() == Unknown)* {. The *getNodeChildren()*, *moveNodeToChild* and *moveNodeToParent* methods can be implemented as in Frame 3.

In AD the weight of a node is computed with the sum of the weights of its children adding its own individual weight. In order to ensure that we calculate the weights of the nodes in linear time, they are calculated only when the node is completed. Note that when a node is completed, its children are also completed and thus no more nodes can be added. Therefore, in trees where nodes are not completed, the weights of some descendants of the root have not been calculated yet and, hence, the weight of the root node cannot be determined. Consequently, the *calculateWeight* method should also be modified. The new version of this method in Frame 4 uses the *getNodeChildren*, *moveNodeToChild* and *moveNodeToParent* methods to transform into children of the root node all those completed nodes in the MET whose parent is not completed. In this way, when the search strategy is used again, the weight of the root will be updated as new nodes become completed.

In this architecture, as in the standard architecture, Lines (24) to (26) of Frame 1 could also be moved to Line (9) using the *numChildren == 0* condition instead of *indexChild == -1* and removing the *this.node.getState() == Undefined* condition in line (16). Here again, these changes make the algorithm to avoid unnecessary checks.

```
private Node parent = null;
private List<Node> children = new ArrayList<Node>();

private int getNodeChildren()
begin
(1)  this.children.clear();
(2)  this.parent = this.node;
(3)  int numChildren = this.getNodeChildren(0);
(4)  this.node = this.parent;
(5)  return numChildren;
end

private int getNodeChildren(int numChildren)
begin
(1)  for (Node child : this.node.children)
(2)      if (child.isCompleted())
(3)          this.children.add(numChildren++, child);
(4)      else {
(5)          this.node = child;
(6)          numChildren = this.getNodeChildren(numChildren);
(7)      }
(8)  return numChildren;
end

private void moveNodeToChild(int index)
begin
(1)  this.node = this.children.get(index);
end

private void moveNodeToParent()
begin
(1)  this.node = this.parent;
end

private double calculateWeight()
begin
(1)  double weight = 0;
(2)  int numChildren = this.getNodeChildren();
(3)  for (int index = 0; index < numChildren; index++) {
(4)      this.moveNodeToChild(index);
(5)      weight += this.node.getWeight();
(6)      this.moveNodeToParent();
(7)  }
(8)  if (this.node.isCompleted() && this.node.getState() != Wrong)
(9)      weight += this.node.getIndividualWeight();
(10) return weight;
end
```

Frame 4: OptimalDivideQuery class (code for the dynamic MET architecture)

6.4 Divide by Queries

6.4.1 Introduction

The search strategy used to decide which is the best node of the Marked Execution Tree (MET) to be asked about is crucial for the performance of Algorithmic Debugging (AD). After the definition of AD, there have been a lot of research to define new search strategies that reduce the amount of questions [95].

In practice, the Divide & Query strategy by Hirunkitti and Hogger [44] needs to perform less questions than other search strategies to find a bug, and this is why it has been implemented in almost all algorithmic debuggers (see, e.g., [85, 10, 24, 88, 67, 13, 87, 46]). However, in Section 6.1.3 we showed that *Divide & Query* is not an optimal strategy because it does not take into consideration the structure of the tree. In fact, the own name of the technique clearly states that the objective of the technique is to find the node that divides the tree into two parts with the same amount of nodes (Divide), and then ask about that node (Query). However, not taking into account the amount of queries that will be performed is the main mistake of *Divide & Query*, and this is why it cannot be an optimal strategy ever. In this section we introduce the *Divide by Queries* strategy, which not only takes into account the queries done, but they are what is actually considered to divide the tree. To explain this strategy, we show the beginnings that it should fulfill, and we provide an approximation [49]. We should make clear that, as demonstrated in [57], the problem we want to solve is a NP-hard problem, and thus we only present methods to approximate the solution.

The rest of the section is structured as follows: In Section 6.4.2 we show that the problem of obtaining such search strategy is decidable. Section 6.4.3 shows how we face the problem and introduces sequences of questions. Finally, Section 6.4.4 explains which of the sequences are optimal and shows how to obtain them.

6.4.2 Decidability

In this section we show the beginnings that an AD search strategy should have to be considered optimal. First of all, we show that the problem that should solve a search strategy is decidable by showing a naive approximation of the algorithm that allows for selecting an optimal node.

Theorem 6.4.1 (Decidability) *Given a MET, finding all optimal nodes is a decidable problem.*

Proof. We face this proof by showing that there exists at least one finite method to find all possible optimal nodes. Firstly, we know that the size of the MET is finite because the question of the root can only be completed when the execution of the program has finished, and therefore, the number of subcomputations—and hence also of nodes—is finite [47]. Because of being a finite tree, we know (by Algorithm 1) that any sequence of questions made by the debugger (without taking into account the search strategy used) is also finite, because in the worst case we can ask as maximum about all the nodes of the tree. Therefore, the number of possible sequences is also finite. This guaranties that it is always possible to calculate all possible sequences and keep afterward only the best ones according with Definition 6.1.1. The optimal nodes would be the first nodes of the selected sequences. □

Although the method presented in Theorem 6.4.1 is effective, it is also computationally expensive because it implies calculating all possible sequences of questions. In the next sections we show a more efficient approximation to find these optimal nodes.

6.4.3 Valid sequence of questions

For clarity reasons, from here on when we mention a sequence of questions of a node, this sequence will assume that the own node is marked as Wrong and it will be composed of a set of nodes that, after being asked about, will allow for determining whether the node is buggy or not.

According with Definition 6.1.1, a sequence of questions is optimal (is what an optimal search strategy would calculate) when the sum of the number of questions produced by the debugger is minimum assuming that the buggy node can be any node of the tree. Therefore, as previously explained in the proof of Theorem 6.4.1, a method to calculate the optimal sequences is to calculate all possible sequences and afterwards to select the bests of them. However, not all possible sequences are valid, and therefore many of them can be omitted.

Definition 6.4.2 (Valid sequence of questions of a node, SP_n) *Let $T = (N, E, M)$ be a MET whose root is $n \in N$. A valid sequence of questions $sp_n = [n_1, \ldots, n_m]$ of n must fulfill:*

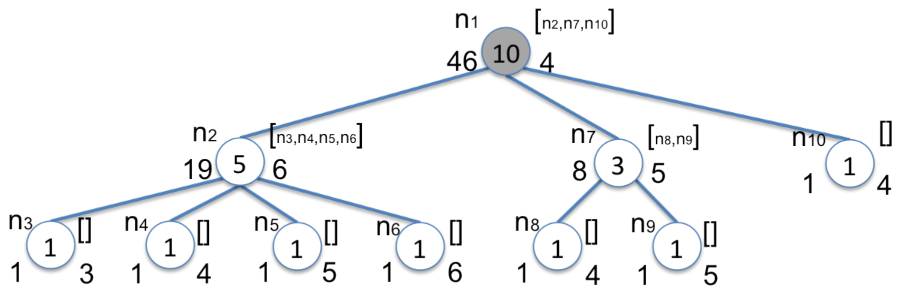
1. $\forall n_i \in sp_n, (n \rightarrow n_i) \in E^+$
2. $\forall n_i, n_j \in sp_n, 1 \leq i < j \leq m, (n_i \rightarrow n_j) \notin E^*$
3. $N \backslash \{n_j \mid (n_i \rightarrow n_j) \in E^* \wedge n_i \in sp_n\} = \{n\}$

We call SP_n the set of all valid sequences of questions.

Roughly speaking, the valid sequences of a n node are all sequences of not repeated nodes that (1) all nodes are descendants of n, (2) a node in the sequence cannot be descendant of a previous node in the sequence, and (3) after pruning all subtrees whose roots are the nodes of the sequence, the n node ends without descendants. Note that, according with the definition, all nodes of a tree can have their own sequence of questions.

In the following example we show that if we label each node of the MET with a valid sequence of questions, then it is possible to know how many questions are necessary to find a buggy node in the MET.

Example 6.4.3 *Consider the next tree formed by two subtrees of depth 2 and one subtree of depth 1.*



Besides the weight and the identifier, each node is labelled with a valid sequence of questions in the top right corner, and in the bottom right corner with the number of questions needed to find a bug in this node.

There exist a lot of possible sequence of questions for the n_1 node, (e.g., $[n_2, n_7, n_{10}]$, $[n_2, n_{10}, n_7]$, $[n_3, n_2, n_7, n_{10}]$, etc.). Using the sequence of the figure ($[n_2, n_7, n_{10}]$) and taking into account that the first node to be asked about is always the root,[2] *we can determine the number of questions needed to*

[2]Imposing that the first node to be asked about is always the root is not a limitation of the method. Note that, in order to find the optimal sequence of the root without starting to ask about it, we can always add a new node as parent of the root and use the method proposed here with this new tree. The obtained sequence would be the optimal sequence of the original tree without starting to ask about its root (i.e., removing the first question).

find a bug wherever it is. For example, to find a bug in the n_1 node the debugger would ask 4 questions (n_1, n_2, n_7, n_{10}). In a similar way, $q_{n_4} = 4$ (n_1, n_2, n_3, n_4).

Once we know the number of questions needed to find a bug for each node (from here on referenced as q_n), it is possible to obtain the number of questions of the whole MET (from here on referenced as Q_n). This number is shown in the previous example in the bottom left corner of each node. Note for instance that the root has a total number of question of 46 ($Q_{n_1} = 46$), the n_2 node has 19 ($Q_{n_2} = 19$), and each leaf has 1 question (this node itself). In some cases it is easy to calculate Q_n. For instance, a moment of thought would convince the reader that the next values for Q_{n_1} using $[n_2, n_7, n_{10}]$, $[n_2, n_{10}, n_7]$, $[n_7, n_2, n_{10}]$ are correct:

$$\begin{array}{lll}
[n_2, n_7, n_{10}] & \rightarrow Q_{n_1} & = (Q_{n_2} + w_{n_2}) + (Q_{n_7} + 2 * w_{n_7}) + (Q_{n_{10}} + 3 * w_{n_{10}}) + (4) \\
 & & = (19 + 5) + (8 + 6) + (1 + 3) + (4) = 46 \\
[n_2, n_{10}, n_7] & \rightarrow Q_{n_1} & = (Q_{n_2} + w_{n_2}) + (Q_{n_{10}} + 2 * w_{n_{10}}) + (Q_{n_7} + 3 * w_{n_7}) + (4) \\
 & & = (19 + 5) + (1 + 2) + (8 + 9) + (4) = 48 \\
[n_7, n_2, n_{10}] & \rightarrow Q_{n_1} & = (Q_{n_7} + w_{n_7}) + (Q_{n_2} + 2 * w_{n_2}) + (Q_{n_{10}} + 3 * w_{n_{10}}) + (4) \\
 & & = (8 + 3) + (19 + 10) + (1 + 3) + (4) = 48
\end{array}$$

Because the aim is to minimize Q_n, we can conclude that the optimal sequence of the n_1 node is $[n_2, n_7, n_{10}]$.

In summary, all collected and labelled information can be shown in Figure 6.12.
In the rest of the section we show a technique to calculate Q_n for any node of the MET. We start by defining two base cases:

- Tree of depth 1 (n is a leaf): $Q_n = 1$
- Tree of depth 2 (n has m children): $Q_n = (\sum_{i=2}^{m+1} i) + m + 1$

These base cases are associated with the following trees:

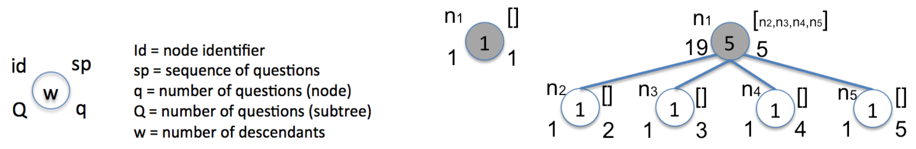

id
sp
w
Q
q
Id = node identifier
sp = sequence of questions
q = number of questions (node)
Q = number of questions (subtree)
w = number of descendants
n1
[]
1
1
1
n1
[n2,n3,n4,n5]
19
5
5
n2
[]
1
1
2
n3
[]
1
1
3
n4
[]
1
1
4
n5
[]
1
1
5

Figure 6.12: Divide by Queries base cases

The base case 1 (the tree in the middle of Figure 6.12) only has one node and therefore, because we start asking about it, in only one question we can find the bug. In the base case 2 (the tree on the right of Figure 6.12), the root is the first node to be asked about, hence all nodes have accumulated this question, and then the children are asked from left to right. If the buggy node is the root, then the debugger needs to ask about and discard all its children, hence the root has a q_n of 5.

In order to calculate Q_n for trees whose depth is 3 or more, a more sophisticated process is needed. Luckily, this process is compositional (i.e., Q_n of a n node can be obtained from the information previously calculated in its descendants) and therefore, it can be obtained while the MET is being generated.

6.4.4 Optimal sequence of questions

In the previous section we saw that the cost Q_n of a node n only depends on its own tree. This number can be used to indicate the amount of questions needed to find a bug in this tree when the debugger starts to ask about the root node. In this section we want to find which is the sequence $sq_n \in SP_n$ that minimizes Q_n for that node. Algorithm 25 obtains this sequence.

Algorithm 25 Calculate optimal sp_n

```
Input: A MET $T = (N, E, M)$ and a node $n \in N$
Output: $(sp_n, Q_n)$
Preconditions: $n.depth$ returns the depth of the $n$ node

begin
 1) if $(n.depth = 1)$
 2)     $spOptimal = []$
 3)     $QOptimal = 1$
 4) else if $(n.depth = 2)$
 5)     $spOptimal = [n_a, ..., n_z] \mid n_a, ..., n_z \in N \wedge (n \rightarrow n_a), ..., (n \rightarrow n_z) \in E$
 6)     $QOptimal = (\sum_{i=2}^{m} i) + m \mid m = 1 + Card(spOptimal)$
 7) else
 8)     $spOptimal = sp_n \in SP_n \mid \nexists sp'_n \in SP_n, calculateQ(T, n, sp_n) > calculateQ(T, n, sp'_n)$
 9)     $QOptimal = calculateQ(T, n, spOptimal)$
10) end if
11) return $(spOptimal, QOptimal)$
end
```

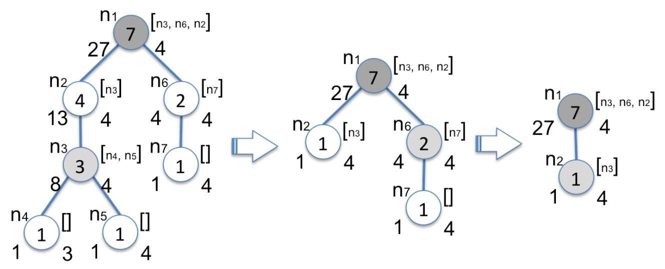
As the MET is being completed, Algorithm 25 can use it to generate (at the same time) the optimal sequence of questions of each generated node. To do this, the algorithm needs to be executed for each completed node (i.e., this cannot have more descendants), so each descendant of the new generated node will always have its optimal sequence. This algorithm has into account both base cases for trees with depth 1 (Lines 2 and 3), and 2 (Lines 5 and 6). If the tree would have a depth of 3 or more, the best sequence is selected from the valid sequences of the node by comparing the Q_n obtained for each valid sequence (Line 8). In order to calculate Q_n for each sequence, we use Algorithm 26.

Algorithm 26 Calculate Q_n from a given sequence of questions

```
Input: A MET $T = (N, E, M)$, a node $n \in N$ and a sequence of questions $sp_n \in SP_n$
Output: $Q_n$
Initialization: $questions = 0$
                $questAcum = 0$

begin
 1) while $(\{n' \mid (n \rightarrow n') \in E^*\} \neq \{n\})$
 2)     $node = sp_n[questAcum]$
 3)     $questAcum = questAcum + 1$
 4)     $questions = questions + (Q_{node} + questAcum * w_{node})$
 5)     $T = adjustNodesInPath(T, n, node)$
 6) end while
 7) $questions = questions + (1 + questAcum)$
 8) return $questions$
end
```

Algorithm 26, after the evaluation of each node of the sequence *node* (Line 4), invokes a method to adjust the $Q_{n'}$ cost of each node n' in the path from n to *node*. (Line 5). This is because the $Q_{n'}$ cost of each node between n and *node* was calculated using the subtree whose root is *node*. However, now, if the search strategy asks about these nodes while using the sequence, these subtrees would have been pruned. Algorithm 27 is in charge of adjusting the $Q_{n'}$ value for each of these nodes.

Example 6.4.4 *Consider the next tree (left) of depth 4, where we want to obtain the cost of* Q_{n_1} *associated with the* $[n_3, n_6, n_2]$ *sequence.*



Algorithm 26 evaluates along the first iteration the n_3 node. After that, Algorithm 27 prunes the subtree of the n_3 node and recalculates the value of Q_{n_2} and w_{n_2}. As it can be seeing, when Algorithm 26 evaluates n_2 along the third iteration (the tree on the right), this node should have a Q_{n_2} associated with the current structure of the MET.

Algorithm 27 Adjust nodes in the path

```
Input: A MET $T = (N, E, M)$ and two nodes $n, n' \in N \mid n \neq n' \wedge (n \to n') \in E^*$
Output: A MET $T' = (N', E', M')$

begin
  1) $O = \{n'' \in N \mid (n' \to n'') \in E^*\}$
  2) $N = N \backslash O$
  3) $n' = n'' \mid (n'' \to n') \in E$
  4) while $(n' \neq n)$
  5)    $(\_, Q_{n'}) = calculateOptimalSp(T, n')$
  6)    $w_{n'} = w_{n'} - |O|$
  7)    $n' = n'' \mid (n'' \to n') \in E$
  8) end while
  9) return $T$
end
```

Algorithm 27 prunes the already evaluated subtree (Lines 1 and 2) and besides calculating the new Q_n value for the nodes in the path (Line 5), it also updates their weights (Line 6) by subtracting the number of removed nodes from their previous weights.

Chapter 7

Implementations

In this chapter we explain the current state of the Declarative Debugger for Java (DDJ). First of all we show all the features implemented into the algorithmic debugger, and later we explain how this debugger has been integrated into Eclipse to work together with two other debugging techniques in such a way that they work together to help the user to find the bug.

7.1 Declarative Debugger for Java (DDJ)

7.1.1 Introduction

Debugging is one of the most important tasks in the software development process. Unfortunately, the efforts of the scientific community in producing usable and scalable debuggers have been historically low. One important example is the lack of an algorithmic debugger for Java. The debugging of Java programs has been traditionally done with the use of breakpoints that allow us to execute the program step by step, and inspect computations (manually) at a given point. However, the adaptation of semi-automatic debugging techniques such as Algorithmic Debugging (AD) has not been successfully done, and neither Sun Java Studio Creator [100], Borland JBuilder [6], NetBeans [76], JCreator [99], nor Eclipse [25] implement an algorithmic debugger.

To the best of our knowledge there has been only one attempt (the algorithmic debugger JDD [38]) of implementing an algorithmic debugger for Java. Other debuggers exist that incorporate declarative aspects such as the Eclipse plugin JavaDD [36] or the Oracle JDeveloper's declarative debugger [35] however, they are not able to automatically produce questions and to control a search to automatically find the bug. This means that they lack current search strategies for AD implemented in standard algorithmic debuggers of declarative languages such as Haskell (Hat-Delta [24]) or Toy (DDT [10]).

The main drawback of JDD is that it suffers from important scalability problems: the internal data structure needed for AD—the so called *Execution Tree* (ET)—is huge (indeed in gigabytes) and it does not usually fit in main memory. Even with the use of a database, the construction of the ET is costly (e.g., minutes). This is the main cause of the lack of algorithmic debuggers for imperative and object-oriented languages such as Java.

For instance, we conducted some experiments to measure the time needed by JDD to start a debugging session[1] with a collection of medium/large benchmarks. Results are shown in column `JDD` of

[1]These times correspond to the execution of the program, the production of the ET and its storage in a database.

Table 7.1. Note that, in order to generate the ET, JDD needs some minutes, thus the debugging session cannot start before this time.

Benchmark	JDD	DDJ
argparser	22 s.	407 ms.
cglib	230 s.	719 ms.
kxml2	1318 s.	1844 ms.
javassist	556 s.	844 ms.
jtstcase	1913 s.	1531 ms.
HTMLcleaner	4828 s.	609 ms.

Table 7.1: Benchmark run-time results evaluating JDD and DDJ

In this section we present the *Declarative Debugger for Java* (DDJ) [46], a new implementation based on the old JDD but with a completely new architecture that incorporates many new features that make the debugging process scalable. In particular, the `DDJ` column of Table 7.1 shows the time needed to start a debugging session with DDJ.[2]

7.1.2 Tool description

This section describes the main features of DDJ, and it explains why its new architecture allows for scaling up in time and memory.

Architecture

The architecture of DDJ is new not only for Java, but for all paradigms. DDJ is the first algorithmic debugger that allows for debugging a program at the same time that the ET is being generated, and it is the first implementation that automatically improves the ETs to reduce the number of asked questions thanks to several techniques presented along this thesis such as Balancing, Loop Expansion and Tree Compression.

The architecture of DDJ is summarized in Figure 7.1. It uses the *Java Platform Debugger Architecture* (JPDA) [72] to generate the ET. This architecture uses the *Java Virtual Machine Tools Interface*, a native interface that helps to inspect the state and to control the execution of applications running in the *Java Virtual Machine* (JVM). Therefore, using JPDA the debugger can execute the source code and inspect the state of the memory at any time. In order to solve the memory scalability problem, DDJ uses a database to store the ET. Therefore, it can work with ETs of any size. Moreover, to avoid showing all the ET to the user, only a part of the ET is shown in the GUI (e.g., the part associated to the current question); the user can configure the amount of ET nodes shown in the GUI. In order to solve the time scalability problem, DDJ implements an algorithm that allows search strategies to work with uncompleted ET nodes. This means that the debugger can start the debugging session when a subcomputation has been executed (i.e., when the ET has at least one completed node), but the program is still running, and thus, the ET is incomplete.

The architecture is composed of three tiers that modularize the processes and limit the memory used in the graphical memory (presentation tier), in main memory (logic tier) and in secondary memory (persistence tier).

All the components of the architecture are described bellow:

[2]These times correspond to the execution of the program up to the generation of the first ET node and its storage in a database.

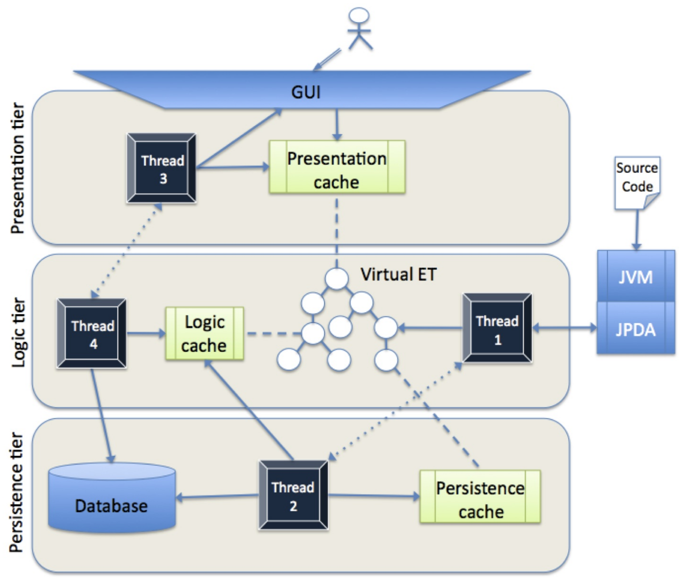


Figure 7.1: Architecture of DDJ

GUI: The GUI is independent of the other components. It can be adapted to interact with other oracles apart from the user.

Database: The database stores the complete ET and it is independent of the architecture. The connection is done via JDBC, thus any database could be used. The current distribution uses MySQL by default.

JPDA / JVM: JPDA is used to control the execution of the source code in the JVM in order to produce the ET. The architecture only communicates with this component via thread 1; therefore, the debugger could be easily adapted to another language (e.g., C++) by only changing thread 1.

Virtual ET: A copy of the ET in the database is always in main memory. However, this copy called "virtual ET" only contains the structure of the ET, being the nodes empty. When it is required, the information of a cluster of nodes is loaded from the database to the virtual ET. This allows us to control how much memory is used by the virtual ET.

Presentation Cache: It contains the subset of the ET shown in the GUI.

Logic Cache: It contains the subset of the ET stored in the virtual ET.

Persistence Cache: It contains a subset of nodes computed by JPDA that will be stored in the database with a single transaction. This clustering mechanism allows for reducing the number of accesses to the database.

Thread 1: It constructs the virtual ET.

Thread 2: It stores the ET in the database.

Thread 3: It interacts with the user.

Thread 4: It controls which nodes of the ET are stored (also) in the virtual ET. It implements all the AD search strategies.

Functionality

An enumeration of the main features and functionalities of DDJ follows:

1. *AD search strategies.* The chosen search strategy strongly influences the performance of AD [95]. DDJ is the algorithmic debugger that currently implements more search strategies, and moreover, it allows the user to change the search strategy during a debugging session. Currently, 13 search strategies are implemented: Top down [2], Single stepping [92], Heaviest first [5], More rules first

[94], Less YES first [95], both Shapiro's [92] and Hirunkitti's [44] Divide & query, Divide by rules & query [94], Divide by YES & Query [95], the three Hat Delta heuristics [24] and our Optimal Divide & Query version [48, 50].

2. *GUI.* Many algorithmic debuggers (e.g., Hat or Buddha) lack a GUI because they are based on a semi-automatic search. However, the user can provide useful information to the debugger if they are allowed to freely (manually) explore the ET. This allows them, for instance, to select a subtree of the ET (e.g., the suspicious one or the one associated to the last changes, etc.) instead of selecting always the whole ET. It also allows them to change the state of some nodes and thus avoiding the exploration of correct parts of the ET.

   Clearly, having a graphical (and interactively explorable) ET can speed up the debugging session. However, this comes at a cost: too much information shown to the user can lead to confusion. To solve this situation, DDJ provides a clustering mechanism (based on the use of the presentation cache) that permits to load the part of the ET that (i) is required by the user, or (ii) is required by the search strategy being used. This mechanism automatically ensures that the current node is shown in the GUI with a number of directly related nodes (ancestors, descentants, siblings, etc.).

3. *Portability.* The architecture of DDJ is similar to the architecture of a compiler. There is a front-end that depends on the source language, and a back-end that is independent of the source language and only depends on the intermediate representation, which in this case is the ET. This means that the GUI, the database, the search strategies, and all the components of the back-end are mostly independent of the language that is being debugged. Therefore, if we change the front-end, e.g., to work with C++, the debugger could debug C++ programs by only changing the implementation of thread 1.

4. *Uncompleted ETs (Section 5.1).* Another feature that makes DDJ different from the other debuggers is that DDJ is able to work with uncompleted ETs. This feature does not only speed up the debugging session but it also allows for making AD scalable. Traditional AD is based on two sequential phases: producing an ET and exploring the ET. Contrarily, in DDJ both phases can be overlapped so that the user is able (if desired) to start the debugging session long before the ET is completed, avoiding to waste time (see Table 7.1), given that the generation of the ET is the bottle neck of AD. This feature required the reimplementation and adaptation of the debugging search strategies to work with uncompleted ETs [47].

5. *Balancing technique (Section 5.2).* DDJ is the first implementation of a novel technique [55] that allows for reducing the number of asked questions performed to the user during the debugging session. This technique is able to automatically balance ETs by a transformation that collapses some nodes and introduces new ones. The result is an ET that is often bigger than the original, but it is more efficient when it is explored by the search strategies.

6. *Loop Expansion technique (Section 5.4).* DDJ also includes a new technique [56] that transforms all iterative loops into recursive methods. This allows for reducing the granularity of the bugs that would be found by the debugger (i.e., the source code reported as buggy is smaller). Before the definition of this technique, all detected bugs were always associated with a method. This technique allows now for indicating a loop inside the method as the responsible of a bug, as well as discarding it.

7. *Tree Compression technique (Section 5.5).* This technique has been created in [24] and it is implemented in some algorithmic debuggers. However, until now, there did not exist a mechanism to determine whether this technique should be applied (as discussed in Section 5.5, tree compression

can speed up, but also slow down, the debugging session). In DDJ we introduce this mechanism [56], which improves the original version, and we provide to the user both the original and the improved one.

8. *Optimal Divide & Query strategy (Section 6.2).* Search strategies are a fundamental part of any algorithmic debuggers. It determines which the next node to be asked about is. In DDJ we have implemented an improved version [48, 50, 51] of the previously best search strategy in the literature.

7.1.3 Usage scenario

This section describes a typical usage scenario of DDJ. We describe a scenario in which the user decides to use the guided search for a bug. In this case, the debugger uses a search strategy to search for the error, and the debugging session is semi-automatic because the user only has to answer questions.

A summary of the steps described in the usage scenario is shown in Table 7.2.

`Steps`	`Output`	`Comments`
`1.- Load a program`	The ET is shown in the GUI	Select the .class file and dependencies (if any), and provide the arguments (if any)
`2.- Select a search strategy`	The GUI shows a question	It is also possible to freely explore the ET and debug the program manually
`3.- Answer questions`	The ET is pruned in the GUI and a new question is shown	Possible answers: correct, wrong, I don't know, trusted
(...)	(...)	Step 3 is repeated until the bug is found
`4.- Locate the bug`	A bug is highlighted in the source code	

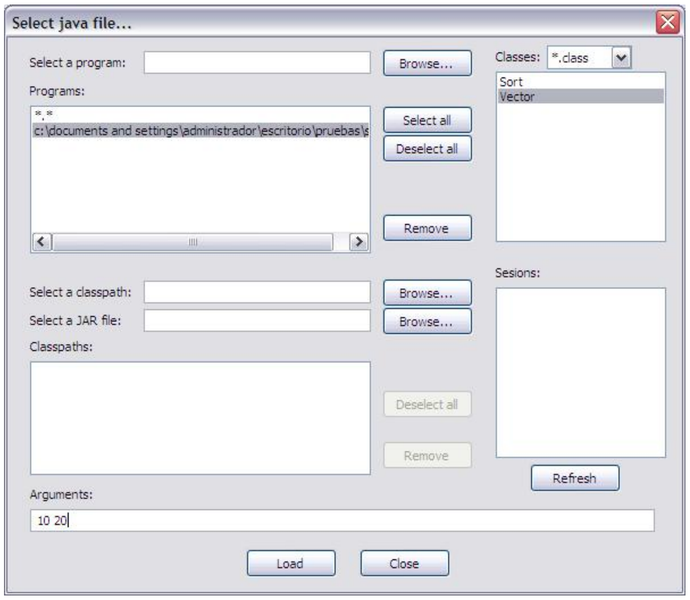
Table 7.2: Usage scenario with DDJ

Step 1: The first step in a debugging session is to load the buggy program. This can be done with the window shown in Figure 7.2 where we see that the *vector.class* file is selected. This file communicates with other files that implement sorting algorithms such as Quicksort. In addition to the *.class*, it is possible to select the dependencies that the program will use, and the arguments used for the execution of the program. In this example the arguments are 10 and 20.

With this information, DDJ executes the provided program and gradually produces the associated ET. A portion of the ET associated to the loaded program is shown in the main panel of Figure 7.3. Even if the ET is incomplete, the debugging session can start as long as one node has been completed. In this case the debugger will only ask about those nodes that have already been completed, and, therefore, the user is able to determine whether the computation of the node is correct or not.

Step 2: Once a program has been loaded, the debugging session can start either manually or guided by the debugger. During a debugging session, at any point, the user can change to manually or guided debugging several times. In manual debugging, the user can explore the ET and select wrong computations or discard computations that are correct. It is also possible to mark computations as trusted (e.g., because the code is reused, it is already tested, etc.). When a node is marked as trusted, the debugger automatically searches for all computations associated with the trusted code, and they are automatically discarded (i.e., marked as correct).

In guided debugging, the user only needs to choose which search strategy to use (e.g., Top-Down) and start the debugging session with a single click.



Select a program:

Programs:
.
c:\documents and settings\administrador\escritorio\pruebas\s

Classes: *.class
Sort
Vector

Sesions:

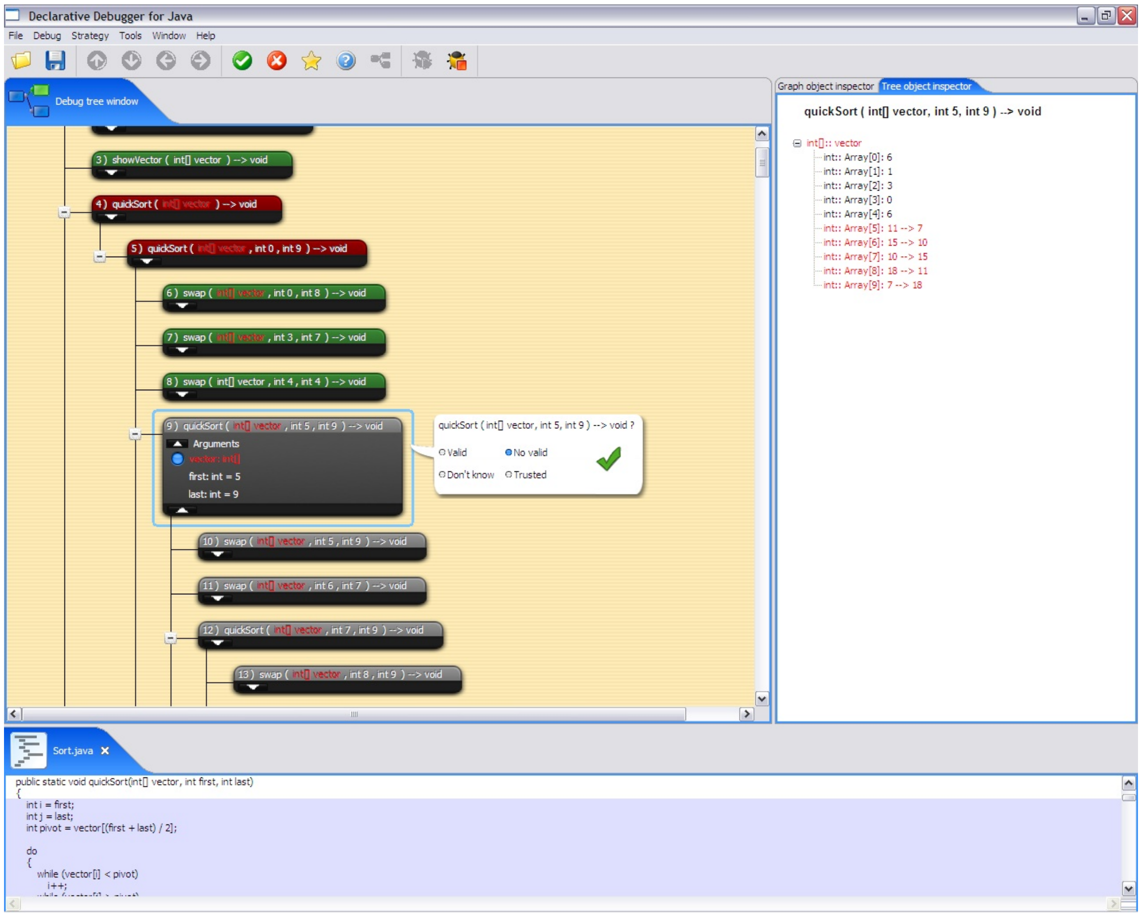
Select a classpath:

Select a JAR file:

Classpaths:

Arguments:
10 20

Figure 7.2: Assistant to load a Java program with DDJ



```
3) showVector ( int[] vector ) --> void
4) quickSort ( int[] vector ) --> void
5) quickSort ( int[] vector , int 0 , int 9 ) --> void
6) swap ( int[] vector , int 0 , int 8 ) --> void
7) swap ( int[] vector , int 3 , int 7 ) --> void
8) swap ( int[] vector , int 4 , int 4 ) --> void
9) quickSort ( int[] vector , int 5 , int 9 ) --> void
   Arguments
   vector: int[]
   first: int = 5
   last: int = 9
10) swap ( int[] vector , int 5 , int 9 ) --> void
11) swap ( int[] vector , int 6 , int 7 ) --> void
12) quickSort ( int[] vector , int 7 , int 9 ) --> void
13) swap ( int[] vector , int 8 , int 9 ) --> void
```

quickSort (int[] vector, int 5, int 9) --> void ?

Valid [✓ No valid]
Don't know Trusted

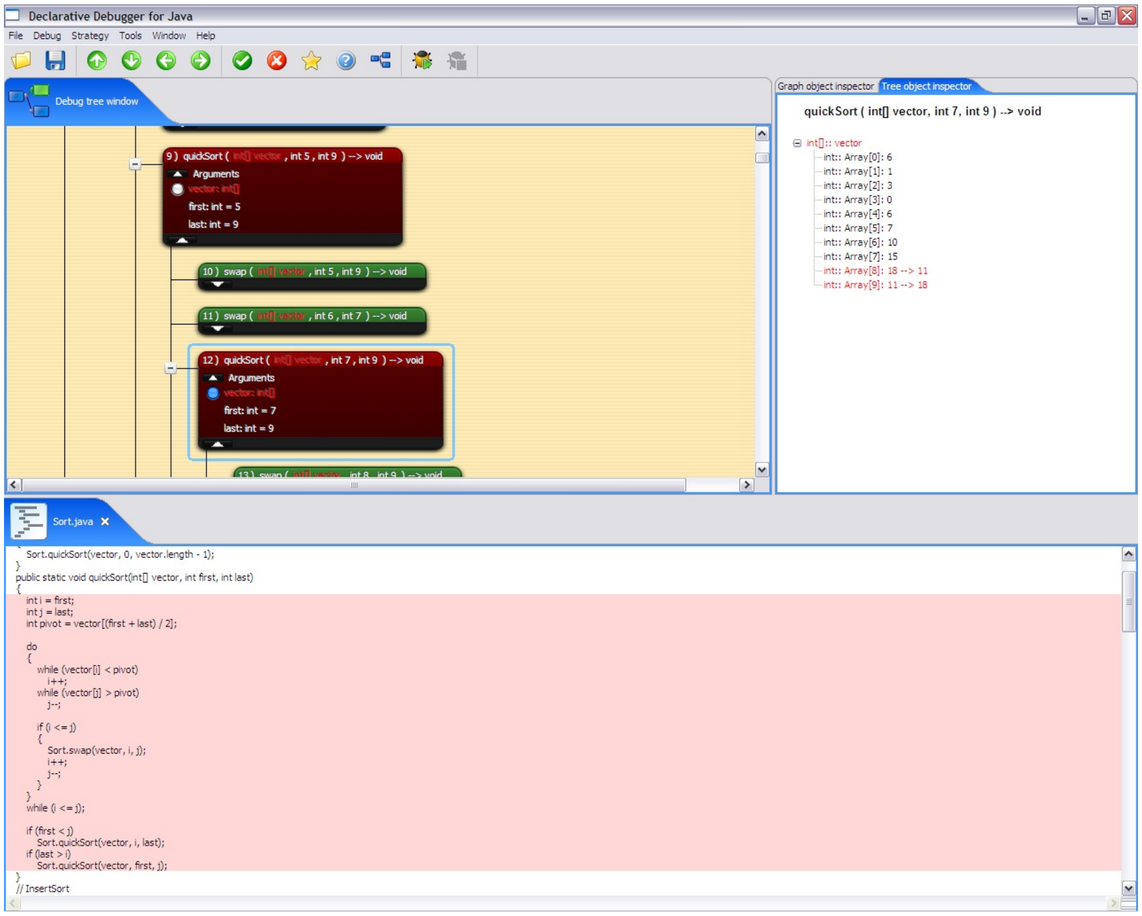
```
quickSort ( int[] vector, int 5, int 9 ) --> void

int[]:: vector
  int:: Array[0]: 6
  int:: Array[1]: 1
  int:: Array[2]: 3
  int:: Array[3]: 0
  int:: Array[4]: 6
  int:: Array[5]: 11 --> 7
  int:: Array[6]: 15 --> 10
  int:: Array[7]: 10 --> 15
  int:: Array[8]: 18 --> 11
  int:: Array[9]: 7 --> 18
```

```
public static void quickSort(int[] vector, int first, int last)
{
   int i = first;
   int j = last;
   int pivot = vector[(first + last) / 2];

   do
   {
      while (vector[i] < pivot)
         i++;
```

Figure 7.3: Question produced by DDJ

Step 3: When the debugging session is guided, it becomes a dialogue between the debugger and the user. The debugger selects a node according to the selected search strategy and asks the user whether the computation of this node is correct. To answer the question, the user can inspect (it is interactively shown in the GUI) all the variables of the program that are in the scope of this computation. These variables are shown with both values before and after the execution, so that the user can inspect the effects of the computation.

If the result of the computation is what the user expected, then the user answers *correct* and the node and its descendants are automatically marked as correct and thus discarded in the search for the bug. In the case that the user is sure that the method appearing in the question of the debugger is well implemented, they can answer *trusted*. As described before, this produces that all the nodes associated with this method are also marked as correct.



```
9) quickSort ( int[] vector , int 5 , int 9 ) --> void
   Arguments
   vector: int[]
   first: int = 5
   last: int = 9
10) swap ( int[] vector , int 5 , int 9 ) --> void
11) swap ( int[] vector , int 6 , int 7 ) --> void
12) quickSort ( int[] vector , int 7 , int 9 ) --> void
   Arguments
   vector: int[]
   first: int = 7
   last: int = 9
13) swap ( int[] vector , int 8 , int 9 ) --> void
```

```
quickSort ( int[] vector, int 7, int 9 ) --> void

int[]:: vector
  int:: Array[0]: 6
  int:: Array[1]: 1
  int:: Array[2]: 3
  int:: Array[3]: 0
  int:: Array[4]: 6
  int:: Array[5]: 7
  int:: Array[6]: 10
  int:: Array[7]: 15
  int:: Array[8]: 18 --> 11
  int:: Array[9]: 11 --> 18
```

```
  Sort.quickSort(vector, 0, vector.length - 1);
}
public static void quickSort(int[] vector, int first, int last)
{
  int i = first;
  int j = last;
  int pivot = vector[(first + last) / 2];

  do
  {
    while (vector[i] < pivot)
      i++;
    while (vector[j] > pivot)
      j--;

    if (i <= j)
    {
      Sort.swap(vector, i, j);
      i++;
      j--;
    }
  }
  while (i <= j);

  if (first < j)
    Sort.quickSort(vector, i, last);
  if (last > i)
    Sort.quickSort(vector, first, j);
}
// InsertSort
```

Figure 7.4: Buggy source code highlighted by DDJ

On the other hand, if the result is not the expected one, then the user answers *wrong*. This means that the execution of this node is wrong, and thus, the error has been generated by itself or by any of its descendant nodes. Finally, if the user does not know what the expected result of the node is, then they can answer *I don't know*, and the debugger will continue the search without taking into account this node and producing a new question.

For instance, in Figure 7.3 we see DDJ with the ET of the execution of the *vector* class. We see in the main panel the part of the ET associated with a Quicksort computation. Some nodes are dark and others are light because the user already answered some of them stating their correctness. At this screenshot, DDJ is asking the question:

```
quickSort(int[] vector, int 5, int 9) --> void
```

where the values of the 'vector' argument are shown in the panel on the right. Here, those variables that changed during the call are in red and their initial and final values are shown. For instance, array[9] had initially the value 7 and it was changed to 18.

The intended meaning of this call to Quicksort is to sort the values at the positions located between position 5 and 9 (both included). In the panel we see that the initial values were [11, 15, 10, 18, 7], and the final values are [7, 10, 15, 11, 18]. Therefore, the user should mark this computation as wrong.

After every user's answer, the debugger uses the provided information to prune the ET and select a new question. This process is repeated until the bug is found.

Step 4: After the dialogue with the user, the debugger identifies one node of the ET as buggy. This node is responsible of the wrong behaviour of the program, and thus the source code associated with it contains the bug. When the bug is found, DDJ shows a message indicating the method that produced the wrong behaviour and allowing the user to see the part of the source code that contains the bug (see Figure 7.5).

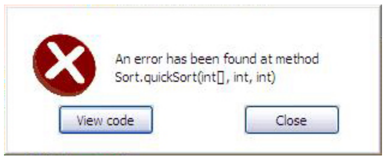

An error has been found at method
Sort.quickSort(int[], int, int)

Figure 7.5: Bug located with DDJ

Figure 7.4 shows the source code panel (at the bottom) where the user can see the source code associated with any node of the ET. In this screenshot, the part of the code associated with the buggy node is highlighted. Observe that during the debugging session, the user does not need to control the execution of the program or use breakpoints and they do not even need to see the source code.

7.1.4 Tool information

DDJ has been completely implemented in Java. It contains about 20400 LOC and it uses SWT for the graphical visualization, which is standard and hence, it can be used in different operative systems. Thanks to JDBC, DDJ can interact with different databases. The current distribution includes both a MySQL and Access databases. The last release of the debugger is distributed in English, Spanish and French.

All described functionalities in this paper are completely implemented in the last stable release. This version is open and publicly available at:

`http://www.dsic.upv.es/~jsilva/DDJ/`

On this website, the interested reader can find installation steps, examples, demonstration videos, and other useful material.

7.2 Hybrid Debugger for Java (HDJ)

7.2.1 Introduction

In this section we introduce a hybrid debugging technique [37] that combines three different techniques, namely, Trace Debugging (TrD), Omniscient Debugging (OD) and Algorithmic Debugging (AD). The combination is done exploiting the strong points of each technique, and counteracting or removing the weak points with their composition. Our method is presented for the programming language Java—our implementation is an Eclipse plugin for Java—but the technique and the architecture of our debugger could be applicable to any other programming language. In summary, the main contributions of this section are the following:

- The design of a new hybrid debugging technique that combines TrD, OD, and AD.
- The integration of the technique on top of the JPDA architecture—which was conceived for tracing, but not for OD or AD—.
- The implementation of the technique as an Eclipse plugin.
- The empirical evaluation of the new architecture that demonstrates the practical scalability of the technique.

The rest of this section is structured as follows: In Section 7.2.2 we analyze the strong and weak points of TrD, OD and AD. Then, in Section 7.2.3 we present our new hybrid debugging technique and explain its architecture. In Section 7.2.4 we describe our implementation, which has been integrated into Eclipse. Finally, the related work is presented in Section 7.2.5.

7.2.2 Debugging techniques

Table 7.3 summarizes the strong and weak points of each technique in HDJ.

Feature	Trace	Omniscient	Algorithmic
Scalability	Very Good	Very bad	Bad
Error granularity	Expression	Expression	Method
Automatized process	Manual	Manual	Semi-automatic
Execution	Forwards	Forwards and backwards	Forwards and backwards
Abstraction level	Low	Low	High

Table 7.3: Comparison of debugging techniques

In our hybrid technique, we want to take advantage of the high abstraction level of AD. We also want to exploit the semi-automatic nature of this technique to speed up bug finding and to avoid errors introduced by the user when searching the bug. However, AD on its own would explore all computations as if they all were suspicious. To avoid this, we want to take advantage of the breakpoints, which provide information to the debugger about what parts of the computation are suspicious for the user (e.g., the last changed code). Hence, we designed our technique to start using the breakpoints introduced by the user, and then automatize the search using AD. Another problem that must be faced is that AD is able to find a buggy method, but not a buggy expression. Therefore, once AD has found a buggy method, we can use OD to further investigate this method in order to find the exact expression that produced the error.

In order to analyze whether this scheme is feasible, we studied the scalability problem of both AD and OD. Operationally, AD and OD are similar. They both record events produced during an execution, and they associate with each event a timestamp. The main difference is that AD only needs to reconstruct the state of the events that correspond to method invocations and method exits (white and black circles in Figure 2.1). Moreover, AD does not need to store information about local variables—only about attributes and global variables—, which is an important difference regarding scalability.

We conducted some experiments to measure the amount of information stored by an algorithmic debugger to produce the ET of a collection of medium/large benchmarks (e.g., an interpreter, a parser, a debugger, etc.) accessible at:

`http://www.dsic.upv.es/~jsilva/DDJ/experiments.html`

Results are shown in Table 7.4.

`Benchmark`	`var. num.`	`ET size`	`ET depth`
`argparser`	8.812	2 MB	7
`cglib`	216.931	200 MB	18
`kxml2`	194.879	85 MB	9
`javassist`	650.314	459 MB	16
`jtstcase`	1.859.043	893 MB	57
`HTMLcleaner`	3.575.513	2909 MB	17

Table 7.4: Benchmark scalability results

The `var. num.` column represents the total amount of variable changes stored in the debugger. The `ET size` column represents the size of the stored information. Observe that the last benchmark needs almost 3 GB. The `ET depth` column is the maximum depth of the ET (e.g., in the `jtstcase` benchmark, there was a stack of 57 activation records during its execution). If we consider that this information does not include local variables, then we can think that the amount of information needed by an omniscient debugger can be huge. Clearly, these numbers show that neither AD nor OD are scalable enough as to be used with the whole program. They should be restricted to a part of the execution. For AD, we propose to restrict its use only to the part of the execution that corresponds to a breakpoint (i.e., the execution of the method where the breakpoint is located). For OD, we propose to restrict its use only to the part of the execution that corresponds to a single method (i.e., the method where AD identified a bug). This proposal is completely in line with the previously discussed ideas: AD will only start in a suspicious area pointed out by a breakpoint, and OD will only be used when a buggy method has been found, and thus the user can trace backwards the incorrect values identified at the end of this method.

7.2.3 Hybrid Debugging

In this section we present our Hybrid Debugger for Java (HDJ) based on the ideas discussed in the previous section. It combines TrD, AD and OD to produce a synergy that exploits the best properties and strong points of each technique.

We start by describing the steps that are followed in a hybrid debugging session. Consider the diagram in Figure 7.6 that summarizes our hybrid debugging method. We see three main blocks that correspond to TrD, AD, and OD. These blocks contain four items that have been numbered; and these items are connected by arrows. Black arrows represent an automatic process (performed by the debugger), whereas white arrows represent a manual process (performed by the user):

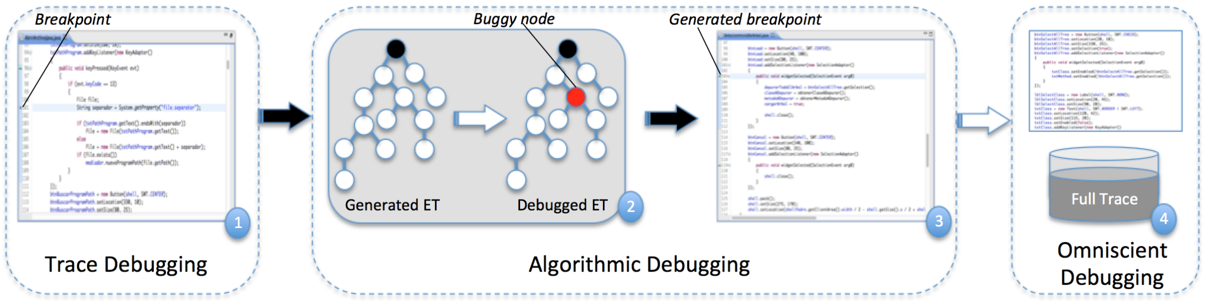


Figure 7.6: Hybrid debugging with HDJ

Trace Debugging. Firstly, after a bug symptom is identified, the user explores the code as usual with the trace debugger and they place a breakpoint b_1 in a suspicious line (probably, inside one of the last modified parts of the code).

Algorithmic Debugging. Secondly, the debugger identifies the method m_1 that contains the b_1 breakpoint, and it generates an ET whose root is associated with m_1. This is completely automatic. Then, the user explores the ET using AD until a buggy node n is found. Note that, according to Theorem 4.2.5, if the m_1 method is wrong, then it is guaranteed that AD will eventually find a buggy node (and thus a buggy method). From n, the AD automatically generates a new breakpoint b_2. b_2 is placed in the definition of the m_2 method associated with n. And, moreover, b_2 is a *conditional* breakpoint that forces the debugger to stop at this definition only when the bug is guaranteed to happen. The condition ensures that all values of the parameters of m_2 are exactly the same as their values in the call to m_2 associated with n.

Example 7.2.1 *Consider a buggy node* $\{x = 0\}\ m(42)\ \{x = 1\}$, *where the definition of method* m, *void* m(*int a*), *is located between lines 176 and 285. Then, the conditional breakpoint generated for it is* $(176, \{x = 0, a = 42\})$. *Alternatively, another conditional breakpoint can also be generated at the end of the method.*

According to Theorem 4.2.4, because the n node is buggy, then the m_2 method contains a bug.

Omniscient Debugging. Thirdly, the debugger acts as an omniscient debugger that explores the m_2 method by reproducing the concrete execution where the bug showed up during AD. The user can explore the method backwards from the final incorrect result of the method. Observe that the OD phase is scalable because it only needs to record the trace of a single method. Note that all method executions performed from this method are known to be correct thanks to the AD phase.

The three described phases produce a debugging technique that takes advantage of all the best properties of each technique. However, one of the most important objectives in our debugger is to avoid a rigid methodology. We want to give the user the freedom of changing from one technique to another at any point. For instance, if the user is using TrD and decides to use OD in a method, they should be able to do it. Similarly, new breakpoints can be inserted at any moment, and AD can be activated when required. The architecture of our tool provides this flexibility that significantly increases the usability of the tool, and we think that it is the most realistic approach for debugging.

Architecture

This section explains the internal architecture of HDJ, and it describes its main features. HDJ is an Eclipse plugin that takes advantage of the debugging capabilities already implemented in Eclipse (i.e., HDJ uses the Eclipse's trace debugger), and it adapts the already existent Declarative Debugger for Java (DDJ, see Section 7.1) [46] to the Eclipse workbench. In fact, all the techniques explained along this thesis are also present in HDJ, the only exception is the *Loop Expansion* technique which could not be adapted. The reason is that the technique is based on adding methods that represent loops, and, currently, Eclipse does not allow for adding new methods to the source code during the debugging sessions. The integration of HDJ into Eclipse is described in Figure 7.7.

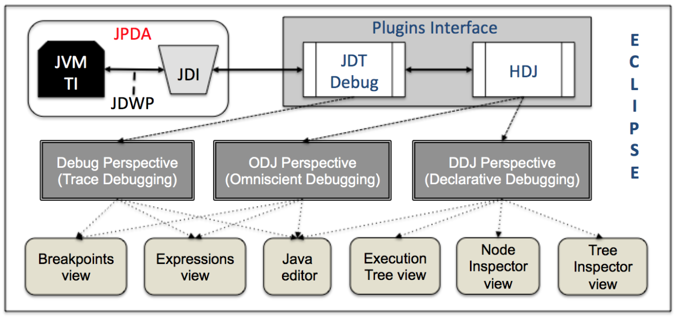


Figure 7.7: Integration of HDJ into Eclipse

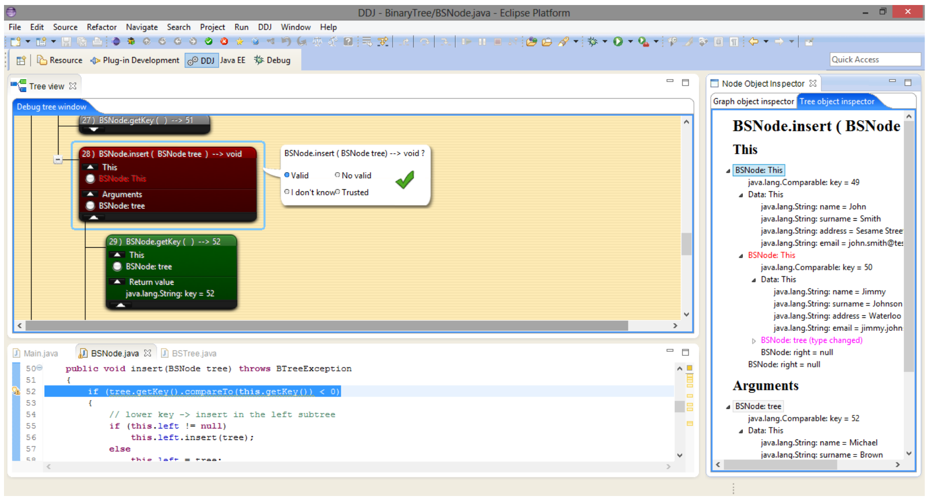
One of the debuggers, the trace debugger, was already implemented by an Eclipse plugin called JDT Debug. The other two debuggers have been implemented in the HDJ plugin. The tool allows the user to switch between three perspectives:

Debug: This perspective allows for performing TrD. It is the standard debugging perspective of Eclipse. It is composed of several views and editors and it offers a wide functionality that includes conditional breakpoints, exception breakpoints, watch points, etc.

ODJ: This perspective allows for performing OD. It contains the `Java editor` and a view that shows the variables in the scope at this point of the execution. The main difference is that ODJ allows for exploring the execution backwards. Internally, it uses a trace of the execution (as the one described in Section 2.2) that is stored in a database.

DDJ: This perspective allows for performing AD. A usage example of this perspective interface is presented in Figure 7.8. In the figure we can see two of its three views and one editor. Firstly, on the left we see the `ET view`, which contains the ET and the questions generated by the debugger. Secondly, on the right we see the `Node inspector`, which shows all the information associated with the selected ET node. This includes the initial context, the executed method and the final context, where changes are highlighted with colours. Thirdly, at the bottom we see the `Java editor`, which contains the source code and the breakpoints. This editor is shared among the three debuggers, and thus, all of them manipulate the same source code, and handle the same user's breakpoints.

One of the important challenges when integrating two new debuggers into Eclipse was to allow all of them to debug the same program at the same time (i.e., giving the user the freedom to change from one debugger to the other in the same debugging session). For this, all of them must have access to the same target source code (e.g., a breakpoint in the target source code should be shared by the debuggers), and use the same target Java Virtual Machine (JVM) and the same execution control over this target JVM. In the figure, this common target JVM is represented with the black box. The Java Virtual Machine Tools Interface (JVM TI) provides both a way to inspect the state and to control the execution running in the target JVM. The debuggers access it through the Java Debug Interface (JDI) whose communication is ruled by the Java Debug Wire Protocol (JDWP). This small architecture to control the JVM is called Java Platform Debugger Architecture (JPDA) [72].



27) BSNode.getKey ( ) --> 51

28) BSNode.insert (BSNode tree) --> void
This
BSNode: This
Arguments
BSNode: tree

BSNode.insert (BSNode tree) --> void ?
[✓ Valid] No valid
I don't know Trusted

29) BSNode.getKey ( ) --> 52
This
BSNode: tree
Return value
java.lang.String: key = 52

```
    public void insert(BSNode tree) throws BTreeException
    {
        if (tree.getKey().compareTo(this.getKey()) < 0)
        {
            // lower key -> insert in the left subtree
            if (this.left != null)
                this.left.insert(tree);
            else
                this.left = tree;
```

BSNode.insert (BSNode

This

```
BSNode: This
    java.lang.Comparable: key = 49
    Data: This
        java.lang.String: name = John
        java.lang.String: surname = Smith
        java.lang.String: address = Sesame Stree
        java.lang.String: email = john.smith@tes
    BSNode: This
        java.lang.Comparable: key = 50
        Data: This
            java.lang.String: name = Jimmy
            java.lang.String: surname = Johnson
            java.lang.String: address = Waterloo
            java.lang.String: email = jimmy.john
        BSNode: tree (type changed)
        BSNode: right = null
    BSNode: right = null
```

Arguments

```
BSNode: tree
    java.lang.Comparable: key = 52
    Data: This
        java.lang.String: name = Michael
        java.lang.String: surname = Brown
```

Figure 7.8: Snapshot of HDJ (DDJ perspective)

The integration of HDJ into Eclipse implies having three different debuggers accessing and controlling the same JVM where the debuggee is being executed. Therefore, our architecture uses two different JVMs that run in parallel and communicate via JPDA. The first JVM is where the debuggee is executed. The second JVM is where the debuggers are executed. It is important to remark that the information of one JVM cannot be directly accessed by the other JVM. Controlling one JVM from the other must be done through JPDA.

A first idea could be to execute the program in the target JVM and stop it when the statement that the user wants to inspect is reached. However, this would imply to re-execute the program once and again every time the user wants to perform a step backwards (i.e., to inspect the previous statement). Obviously, this is a bad strategy, because every time the program is re-executed, the state could change due to, e.g., concurrency, nondeterminism, inputs, etc. Therefore, even if we reach the same statement, it could vary between executions, and the information shown to the user would not be confident. Hence, we need to use some memorization mechanism to store all relevant states of a single execution.

Prior to our current implementation, our first design was conceived in such a way that the JVM of the debugger directly controlled the JVM of the debuggee using communication through JPDA. This implementation had to establish communication between both JVMs after every relevant event. This produced a heavy interaction with a massive message passing that was not scalable even for small programs. Therefore, we designed a second strategy whose key idea is to let the JVM of the debuggee to control itself. More precisely, before executing the debuggee in the JVM, we load a thread in this JVM so that, this thread directs the debugging of the program, thus, avoiding unneeded communication through JPDA. The drawback of this approximation is that to load this thread we first need to add a class and instantiate an object of this class in the first lines of the debugee. Figure 7.9 summarizes this internal architecture of the debugger to control the execution of the debuggee.

The big boxes represent two JVMs. One for the debugger, and one for the debuggee. The debugger has two independent modules that can be executed in parallel: The algorithmic debugger DDJ, and the omniscient debugger ODJ. Each dark box represents a thread. DDJ has four threads: *interface* to control the GUI, *construction* to build the ET, *control* to control and communicate with the debuggee JVM, and *selection* to select the next question. ODJ has two threads: *interface* and *control* that perform similar tasks as in DDJ. In the debuggee, a new thread is executed in parallel with the program. This thread, called *HDJ*, is in charge of collecting all debugging information and storing it in a database. This information is later retrieved by the *control* threads. The *HDJ* thread makes this approach scalable, because it allows for retrieving all the necessary information with a very reduced set of JPDA connections. In the case of OD, the information stored in the database by the *HDJ* thread contains all changes of

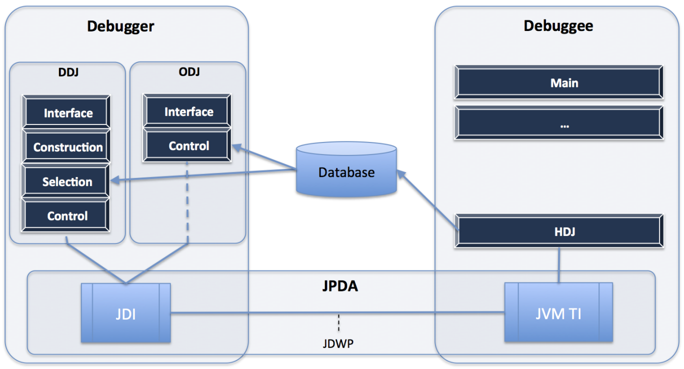
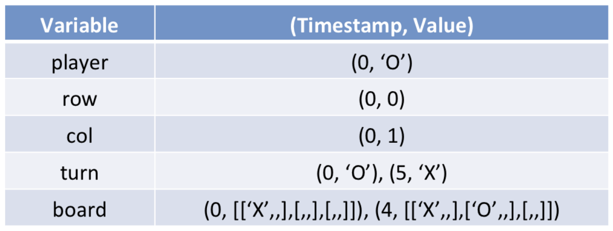


Figure 7.9: Architecture of HDJ

the values of variables that occurred during the execution of the method being debugged.

Example 7.2.2 *Consider again the debugging session in Example 3.0.1. In this debugging session AD determined that the* `mark` *method is buggy, and that the bug shows up with the specific* `game.mark('O',0,1)` *call. With this information, HDJ automatically generates a conditional breakpoint to debug this call. The information stored in the database by the* HDJ *thread for this call is shown in Figure 7.10. Observe that only the variables that changed their value during the execution are stored.*



Variable	(Timestamp, Value)
player	(0, 'O')
row	(0, 0)
col	(0, 1)
turn	(0, 'O'), (5, 'X')
board	(0, [['X',,],[,,],[,,]]), (4, [['X',,],['O',,],[,,]])

Figure 7.10: Information stored in the database by the omniscient debugger

7.2.4 Implementation and empirical evaluation

HDJ has been completely implemented in Java. It contains about 29000 LOC: 19000 LOC correspond to the implementation of the algorithmic debugger (the internal functionality of the algorithmic debugger has been adapted from the DDJ debugger with some extensions that include the communication with JPDA trough JDT Debug, and the perspective GUI), 8300 LOC correspond to the implementation of the omniscient debugger that has been implemented from scratch, and 1700 LOC correspond to the implementation of the own plugin and its integration and communication with Eclipse. The debugger can make use of a database to store the information of the ET and the trace used in OD (if the database is not activated, the ET and the trace are stored in main memory). Thanks to JDBC, HDJ can interact with different databases. The current distribution includes both a MySQL and Access databases. The last release of the debugger is distributed in English, Spanish and French.

All functionalities described in this section are completely implemented in the last stable release. This version is open and publicly available at:

`http://www.dsic.upv.es/~jsilva/HDJ/`

On this website, the interested reader can find installation steps, examples, demonstration videos, and other useful material.

In order to measure the scalability of our technique, we conducted a number of experiments to achieve the time needed by the debugger to start the debugging session. The scalability of TrD is ensured by the own nature of the technique that reexecutes the program up to a breakpoint, and then shows the current state. In fact, we use the Eclipse's standard trace debugger that is scalable no matter where the breakpoint is placed. In the case of AD, scalability could be compromised if the debugger is forced to generate the ET of the whole execution. Even in this case, we are able to ensure scalability:

1. The memory problem is solved with a database. Our debugger never stores the whole ET in main memory. It uses a clustering mechanism to store and load from the database the subtrees of the ET that are dynamically needed by the GUI.

2. The time problem is solved by allowing the debugger to start the debugging session even if the ET is not completely generated (i.e, our debugger is able to debug incomplete ETs while they are being generated) [47] (see Section 5.1).

In the case of OD, we cannot ensure scalability if it is applied to the whole program. For this reason, we limit the application of OD to a single method. This is scalable as demonstrated by our empirical evaluation whose results are shown in Table 7.5.

Benchmark		Execution	Omniscient
Statements	Objects	Time (ms)	Time (ms)
0 - 9 (294)	0 - 1 (96)	5	2060
	2 - 5 (97)	273	3265
	6 - 18 (101)	27	4099
10 - 19 (29)	7 - 18 (10)	51	6527
	19 - 24 (10)	739	7348
	25 - 32 (9)	2062	12379
20 - 57 (10)	25 - 39 (3)	83	3999
	40 - 54 (4)	117	6347
	55 - 100 (3)	176	3757

Table 7.5: Benchmark results for OD

This table summarizes the results obtained for 333 benchmarks. Each benchmark measures the time needed to generate all the information used in OD (the information stored in the database by the *HDJ* thread in Figure 7.9). After this time, the debugger contains the state at any point in the method, and thus, the user can make backwards steps, jump to any point in the method and show the values of the variables at any point. These benchmarks correspond to all methods (333 different methods) executed by the loops2recursion Java library [52] (see Section 5.3) applied over a collection of 25 Java projects. This library automatically transforms all loops in the Java projects to equivalent recursive methods.

All benchmarks have been grouped into three categories according to the number of statements executed in the method (0-9, 10-19, and 20-57). Inside each category, we indicate the number of benchmarks that fall on this category between parentheses. The categories have been divided into subcategories that indicate the number of objects that have changed during the execution of the method (i.e., the number of objects that must be inspected and stored in the database). We have a total of 9 subcategories. Each of them indicates the average time needed to execute the methods in that subcategory (`Execution Time`), and the time needed to generate the information of OD (`Omniscient Time`). All the information is generated between 2 and 12 seconds. The variability between the rows is dependent on the size of the objects that have been changed. Clearly, row 6 has less objects to store than rows 7, 8 and 9, but these objects are bigger, and thus both the execution and omniscient times are higher.

7.2.5 Related work

While a trace debugger is always present in modern development environments, algorithmic debuggers and omniscient debuggers are very unusual due to their previously discussed scalability problems. There exist, however, a few attempts to implement algorithmic debuggers for Java such as the algorithmic debugger JDD [38] and its more recently reimplemented version DDJ [46] (see Section 7.1). Other debuggers exist that incorporate declarative aspects such as the Eclipse plugin JavaDD [36] or the Oracle JDeveloper's declarative debugger [35] however, they are not able to automatically produce questions and to control the search to automatically find the bug. This means that they lack the common search strategies for AD implemented in standard algorithmic debuggers of declarative languages such as Haskell (Hat-Delta [24]) or Toy (DDT [10]). None of these debuggers can work with breakpoints as our debugger does.

The situation is similar in the case of omniscient debuggers. To the best of our knowledge, OmniCore CodeGuide [83] is the only development environment for Java that includes by default an omniscient debugger. Nevertheless, for the sake of scalability, this debugger uses a trace limited to the last few thousands events. Some ad-hoc implementations exist that can work stand-alone or be integrated in commercial environments [45, 86, 60, 73, 61]. Almost all these works focus on how to make OD more scalable [61, 86]. For instance, by reducing the overhead of trace capture as well as the amount of information to store using partial traces that exclude certain trusted classes from the instrumentation process [60]. Other works try to enhance OD, e.g., with causality links [73] that provide the ability to jump from the point a value is observed in a given variable to the point in the past when the value was assigned to that variable. This can certainly be very valuable to resolve the chain of causes and effects that lead to a bug.

There have been several attempts to produce hybrid debuggers that combine different techniques. The debugger ODB [60] combines TrD with OD. It allows the user to debug the program using TrD and start recording the execution for OD when the user prefers. The debugger by Kouh et al. [59] combines AD with TrD. Once the algorithmic debugger has found a buggy method, they continue the search with a trace debugger to explore this method (forwards) step-by-step. This idea is also present in our debugger, but we use OD instead of TrD, and thus we also permit backward steps. The debugger JIVE [33] combines TrD, OD and dynamic slicing. It does not use AD, but allows the user to perform queries to the trace.

To the best of our knowledge, JHyde [43] is the only previous technique that combines TrD, OD and AD. Unfortunately, we have not been able to empirically evaluate this tool (it is not publicly accessible); but considering its architecture, it is highly probable that it suffers from the same scalability problems as any other omniscient debugger. Unlike our solution, their architecture is based on program transformations that instrument the code to store the execution trace in a file as a side effect. First, this instrumentation and the execution of the trace usually takes a lot of time with an industrial program, so that the user has to wait a long time before starting to debug; and second, they store the trace of the whole program, while our scheme only needs the trace of a single method. The common point is that both techniques are implemented as an Eclipse plugin, and they both use the same data structure for OD and AD. This is important to reuse the trace information collected by the debugger. Another important feature implemented by both techniques is the use of a colour vocabulary used in the views. This is very useful to allow the user to quickly see the changes in the state.

Chapter 8

Reformulation of Algorithmic Debugging

In this chapter, we propose a new redefinition [53] of Algorithmic Debugging (AD) in such a way that:

1. It is paradigm- and language-independent, and thus it is reusable by other researchers.
2. It is a conservative generalization of the traditional formulation of AD, in such a way that many previous AD techniques are a particular case of this new formulation.
3. It is formulated in a way that definitions of the data structures, properties, search strategies, and algorithms are specified separately, so that they can be reused and/or concretized in a particular case.
4. It states that the output of an algorithmic debugger should contain dynamic information (i.e., it should not include non-executed code), and,
5. it allows the debugger to ask questions about a code inside a method (and not only about the whole method).

8.1 Some problems identified in current algorithmic debuggers

This chapter somehow summarizes and criticizes our own previous work to make a step forward. We claim that almost all current algorithmic debuggers—at least all that we know, including the most extended, which we compared in [19], and including our own implementations—have fundamental problems that were somehow inherited from the original formulation of AD [92].

In particular, we claim that the original formulation of AD, and most of the later definitions and implementations are obsolete with respect to the last advances in the practical side of AD. For instance, two important problems of the standard definitions of AD are the granularity and the static nature of the located errors (AD reports a whole routine as buggy). We can illustrate these problems observing again the debugging session of Example 3.0.1: The whole `win` method is pointed out as buggy. This is very imprecise especially if `win` were a method with a lot of code. However, AD researchers and developers are used to this behaviour, and they would argue that this is the normal output of any algorithmic debugger. However, from an engineering perspective, this is quite surprising because the analysis performed by the debugger is dynamic by definition (in fact, the whole program is actually executed). Hence, the debugger should know that the last `if` in method `win` of Figure 3.1 is never executed, and thus it should not be reported as buggy. This leads us to our first proposition: The information reported by an algorithmic debugger should be dynamic instead of static. That is, the output of the algorithmic debugger should be the part of the method that has been actually executed to produce the bug, instead of the whole method.

We think that this problem comes from the first implementations of AD and it has been inherited in later theoretical and practical developments. In fact, if we execute this program with the debuggers: Buddha [85], DDT [10], Freja [78], Hat-Delta (and its predecessor Hat-Detect) [23], B.i.O. [8], Mercury's Algorithmic Debugger [68], Münster Curry Debugger [66], Nude [75], DDJ [46], and HDJ [37], they all would output the whole `win` as buggy together with a counterexample that produces the bug (the located buggy node). Unfortunately, none of these debuggers makes further use of the counterexample. An option would be that the debuggers use dynamic program slicing (to be precise, dynamic chopping) [98] to minimize the code shown as buggy.

Traditionally, AD reports a whole method as buggy. To reduce the granularity of the reported errors, new techniques have appeared (see, e.g., [56, 14]) that allow for debugging inside a method. Unfortunately, the standard definition of ET is not prepared for that. In fact, some of the recent transformations defined for AD do not fit in the traditional definition of the data structures used in this discipline. For instance, the *Tree Balancing* technique presented in [55], or the *zooming* technique presented in [14] cannot be represented with standard AD data structures such as the *Evaluation Dependence Tree* [81].

This lack of a common theoretical framework with standard data structures that are powerful enough as to represent recent developments makes researchers to reinvent the wheel once and again. In particular, we have observed that researchers (including ourselves) have produced local and partial formalizations to define their debuggers for a particular language and/or implementation (see, e.g., [16, 55, 56]). These theoretical developments are hardly reusable in other languages, and thus, they only serve as a formal description of their system, or as a means to prove results.

8.2 Paradigm-independent redefinition of Algorithmic Debugging

Some of our last developments for AD cannot be formalized with the standard AD formulation. In a few cases, we just skipped the formalization of our technique and provided an implementation. In other cases we wanted to prove some properties, and thus we formalized (for one specific language, e.g., Java) the part of the system affected by those properties. Other developments were done for other paradigms, e.g., the functional paradigm, and we also formalized a different part of the system with different data structures. We have observed the same behaviour in other researchers, and clearly, this is due to the lack of a standard solution.

We want to provide a definition of AD that is paradigm-independent (i.e., it can be used by either imperative- or declarative- languages). To the best of our knowledge, there does not exist such a formal definition of AD. Hence, in this section we formulate AD in an abstract way. The main generalization of our new formulation is to consider that ET nodes are not necessarily routines as in previous definitions (see, e.g., [81]). Contrarily, we allow ET nodes to contain any piece of code. This permits AD to report any code as buggy, and not only routines, thus potentially reducing the granularity of the reported errors to single expressions.

In the following, we will only call *Execution Tree* to our new definition, and we will call *Routine Tree* (RT) to the traditional definition (that we also formalize in the next sections). Because our new definition is a conservative generalization, the RT is a particular case of the ET as it can be observed in the UML model of Figure 8.1.

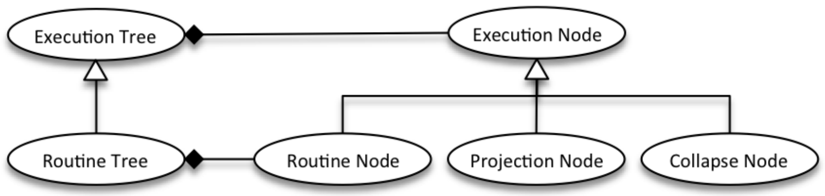


Figure 8.1: UML model representing the structure of the ET

Observe that an execution node can be specialized depending on the piece of code it represents. In particular, we specialize three kinds of execution nodes named *Routine Node*, *Projection Node*, and *Collapse Node*. They correspond to definitions that already exist in the literature (see [31, 55]), but other kinds of nodes could appear in the future.

8.2.1 The Execution Tree

In this section we introduce some notation and formalize the notion of Execution Tree used in the rest of the chapter. We want to keep the discussion and definitions in this section paradigm-independent. Hence, we consider programs as state transition systems.

Definition 8.2.1 (Program) *A* program $P = \{W, I, R, C\}$ *consists of:*

- W*: A set of* states.
- I*: A set of* starting states*, such that* $I \subseteq W$*.*
- R*: A* transition relation*, such that* $R \subseteq W \times W$*.*
- C*: A* source code*, composed of a set of statements.*

Definition 8.2.2 (Computation) *A* computation *is a maximal sequence of states* $s_1, s_2, \ldots$ *such that:*

- s_1 *is a starting state, i.e.,* $s_1 \in I$*.*
- $(s_i, s_{i+1}) \in R$ *for all* $i \geq 1$ *(and* $i \leq n-1$*, if the sequence is of the finite length* n*).*

A finite segment $s_i, s_{i+1}, \ldots, s_j$ *where* $1 \leq i < j \leq n$ *is called a* subcomputation.

In the source code of a program, we consider statements[1] as the basic execution unit. Therefore, in the following, the source code of a program P is a set of statements $st_1, st_2, \ldots, st_n$ that produces the computation $s_0, s_1, \ldots, s_m$ for a given starting state s_0. We cannot provide a specific model of computation if we want to be paradigm-independent, thus we do not define the relation between statements and the transition relation R. This is possible (and convenient) thanks to the abstract nature of AD. In particular, AD only needs an initial state, a code, and a final state to identify bugs. No matter how the code makes the transition from the initial state to the final state. The user will decide whether this transition is correct or not.

Because the considered execution unit is the statement, it is possible to identify a bug in a single statement. This contrasts with traditional AD where routines are the execution units, and thus a whole routine is always reported as buggy.

We also use the notion of *code fragment* of a program P, which refers to any subset of statements in the source code C of P that produces a subcomputation $s_i, \ldots, s_j$ with $0 \leq i < j \leq m$. Code fragments

[1]Note the careful use of the word "statement" to refer to either imperative instructions, declarative expressions, etc.

often represent functions or loops in a program, but they can also represent blocks, single statements, or even function calls together with the whole called function.

Intuitively, not all the statements in a given code c that produces a computation $\mathcal{C}$ are actually executed. Some parts of the code are not needed to produce the computation (e.g., because they are dead code, because some condition does not hold, etc.). The projection of c modulo $\mathcal{C}$ is a subset of c where the unneeded code in c to produce $\mathcal{C}$ has been removed. Projections are often computed with dynamic slicing [98].

Definition 8.2.3 (Code Projection) *Given a code fragment c and a computation $\mathcal{C}_c = s_0, \ldots, s_n$ produced by c from a given initial state s_0, a* projection of c modulo $\mathcal{C}_c$ *is a code fragment that contains the minimum subset of c needed to produce the computation $\mathcal{C}_c$.*

We assume that each state in W is composed of pairs variable-value. The initial and final states, s_i and s_j, describe the effects of a given code fragment c. All three together form a *code behaviour*.

Definition 8.2.4 (Code Behaviour) *Given a code fragment c and a computation $\mathcal{C}_c = s_0, \ldots, s_n$ produced by c from a given initial state s_0, the* code behaviour *of $\mathcal{C}_c$ is a triple $(s_0, \mathcal{P}_{\mathcal{C}_c}(c), s_n)$, where $\mathcal{P}_{\mathcal{C}_c}(c)$ is the projection of c modulo $\mathcal{C}_c$.*

Code behaviour corresponds to the questions asked by the debugger. These questions are along the lines of *Should the code c with the initial state s_0 produce the final state s_n?*, or *Code c produced s_n from s_0, is that correct?* Many previous definitions of AD (see, e.g., [79, 14]) define the code behaviour as the triple (s_0, c, s_n), which corresponds to the execution of a routine c, and usually the debugger only needs to show the call to c instead of showing both the call to c and the own routine c. Definition 8.2.4, however, introduces two important novelties:

- It allows c to be any code fragment, and not only a routine.
- It substitutes c by a projection of c modulo $\mathcal{C}_c$, thus the code associated with a code behaviour only contains the code actually needed to produce that behaviour.

This dynamic notion is much more precise than the usual static notion that considers (the complete code of) a routine.

Definition 8.2.5 (Intended Model) *Given a program $P = \{W, I, R, C\}$, an* intended model *$\mathcal{M}$ for P is a set of tuples $(s_i, \mathcal{P}(c), s_j)$ where $s_i, s_j \in W$ and $\mathcal{P}(c)$ is a projection of a code fragment $c \subseteq C$.*

Each tuple of the form $(s_i, \mathcal{P}(c), s_j)$ specifies that the execution of code $\mathcal{P}(c)$ from state s_i leads to state s_j. Intuitively, an intended model of a program contains the set of code behaviours that are correct with respect to what the programmer had in mind when they programmed these codes. It is used as a reference point against which one can compare computations to determine whether they are correct or wrong.
We are now in a position to define the nodes of an ET.

Definition 8.2.6 (Execution Node) *Let $P = \{W, I, R, C\}$ be a program. Let $\mathcal{C}_c$ be a computation produced by a code fragment $c \subseteq C$. Let $\mathcal{M}$ be an intended model for P. The* execution node *induced by $\mathcal{C}_c$ is a pair ($\mathcal{B}$, $\mathcal{S}$) where:*

1. *$\mathcal{B}$ is the code behaviour of $\mathcal{C}_c$, and*
2. *$\mathcal{S}$ is the state of the node, which can be either:*

- undefined, *or*
- *the correctness of $\mathcal{B}$ with respect to $\mathcal{M}$:* $\begin{cases} \text{correct} & \textit{if } \mathcal{B} \in \mathcal{M} \\ \text{wrong} & \textit{if } \mathcal{B} \notin \mathcal{M} \end{cases}$

Observe that an execution node contains (inside $\mathcal{B}$) the source code $\mathcal{P}_{\mathcal{C}_c}(c)$ responsible of the computation it represents. Hence, if this node is eventually declared as buggy, its associated code is uniquely identified. This definition of execution node is general enough as to represent previous nodes that are used in different techniques. For instance, if the code of the node is a function, it can be represented as a *routine node*. Similarly, *projection nodes* and *collapse nodes*, introduced in [55], are special nodes that agglutinate the code of several other nodes. Clearly, they are also particular cases of our general definition.

In order to properly define ET, we need to define first a relation between execution nodes that specifies the parent-child relation.

Definition 8.2.7 (Execution Nodes Dependency) *Let N be a set of execution nodes. Given an execution node $n_c \in N$ induced by a computation $\mathcal{C}_c$, and an execution node $n_{c'} \in N$ induced by a subcomputation $\mathcal{C}_{c'}$ of $\mathcal{C}_c$, we say that n_c directly depends on $n_{c'}$ (expressed as $n_c \overset{N}{\to} n_{c'}$) if and only if there does not exist an execution node $n_{c''} \in N$ induced by a subcomputation $\mathcal{C}_{c''}$ of $\mathcal{C}_c$, such that $\mathcal{C}_{c'}$ is a subcomputation of $\mathcal{C}_{c''}$.*

Observe that this dependency relation is intransitive, which is needed to define the parent-child relation in a tree. Hence, provided that we have three execution nodes, n_1, n_2, n_3, if $n_1 \overset{N}{\to} n_2 \overset{N}{\to} n_3$ then $n_1 \overset{N}{\not\to} n_3$.

Example 8.2.8 *Given the following program:*

CODE:

```
x++; y++; x=x+y;
```

and the initial state `(x=1,y=2)` *we can generate the following execution nodes (among others):*

ET NODES:

```
              (initial state)      code               (end state)
------------------------------------------------------------------
node 1:       (x=1,y=2)            x++; y++; x=x+y;    (x=5,y=3)
node 2:       (x=1,y=2)            x++; y++;           (x=2,y=3)
node 3:       (x=2,y=3)            x=x+y;              (x=5,y=3)
node 4:       (x=2,y=2)            y++;                (x=2,y=3)
```

with $N = \{$ *node 1, node 2, node 3, node 4* $\}$
we have node 1 $\overset{N}{\to}$ node 2 $\overset{N}{\to}$ node 4 *and* node 1 $\overset{N}{\to}$ node 3

Finally, we define an *Execution Tree*. It essentially represents the execution of a code in a structured way where each node represents a sub-execution of its parent. Formally,

Definition 8.2.9 (Execution Tree) *Let $\mathcal{C}_c$ be a computation produced by a code fragment c. An* Execution Tree *(ET) of $\mathcal{C}_c$ is a tree $T = (N, E)$ where:*

- *$\forall n \in N$, n is the execution node induced by a subcomputation of $\mathcal{C}_c$,*
- *The root of the ET is the execution node induced by $\mathcal{C}_c$,*
- *$\forall (n_1, n_2) \in E$. $n_1 \overset{N}{\to} n_2$.*

This definition is a generalization of the usual call tree (CT), which in turn comes from the refutation trees initially defined for AD in [91, 92]. One important difference between them is that, given a computation $\mathcal{C}_c$ produced by a code fragment c, the CT associated with $\mathcal{C}_c$ is unique because it is only formed of routine nodes. In contrast, there exist different valid ETs associated with $\mathcal{C}_c$ due to the flexibility introduced by the execution nodes (i.e., with routine nodes only one N set is possible, while with execution nodes different N sets are possible). This flexibility of having several possible valid ETs to represent one computation is interesting because it leaves room for transforming the ET and still being an ET. Contrarily, the CT cannot be transformed because it would not be a CT anymore.

Once the ET is built, the debugger traverses the ET asking the oracle about the correctness of the information stored in each node. Using the answers, the debugger identifies a *buggy node* that is associated with a *buggy code* of the program. We can now formally define the notion of buggy node.

Definition 8.2.10 (Buggy Node) *Let $T = (N, E)$ be an ET. A* buggy node *of T is an execution node $n = (\mathcal{B}, \mathcal{S}) \in N$ where:*
(i) $\mathcal{S} = \text{wrong}$*, and*
(ii) $\forall n' = (\mathcal{B}', \mathcal{S}') \in N,\ (n, n') \in E\ \ .\ \ \mathcal{S}' = \text{correct}.$
Moreover, we say that a buggy node n is traceable *if and only if:*
(iii) $\forall n' = (\mathcal{B}', \mathcal{S}') \in N,\ (n', n) \in E^*\ \ .\ \ \mathcal{S}' = \text{wrong}.$

We use E^* to refer to the symmetric and transitive closure of E. This is the usual definition of buggy node (see, e.g., [80]): a wrong node with all its children correct. We also introduce the notion of *traceable*. Roughly, traceable buggy nodes are those buggy nodes that may be directly responsible of the wrong behaviour of the program (their effects are visible in the root of the tree). This property makes them debuggable by all AD search strategies that are variants of Top-Down (see [97]).

Lemma 8.2.11 (Buggy Code) *Let T be an ET with a buggy node $((s, d, s'), \mathcal{S})$ whose children are $((s_1, d_1, s'_1), \mathcal{S}_1)$, $((s_2, d_2, s'_2), \mathcal{S}_2) \ldots ((s_n, d_n, s'_n), \mathcal{S}_n)$. Then, $d \setminus \bigcup_{1 \le i \le n} d_i$ contains a bug.*

Note that we use (s, d, s') meaning $(s, \mathcal{P}_{\mathcal{C}_c}(c), s')$ for some c, and $\setminus$ is the set difference operator.
Proof. Trivial adaptation from the proof by Lloyd [63] for Prolog. □

Lemma 8.2.11 illustrates what (buggy) code should be shown to the user. When a buggy node is detected, the (buggy) code shown to the user is the code of the buggy node minus the code of its children.

8.2.2 The Routine Tree

In this section we formalize the notion of RT used in most AD literature as a particular case of the ET. We call *routine tree* to this specialization of the ET to make explicit its multi-paradigm nature, because routines can refer to functions, procedures, methods, predicates, etc. We first define a *routine node*, which is a specialization of an execution node.

Definition 8.2.12 (Routine Node) *A* routine node *is an execution node $((s_0, \mathcal{P}_{\mathcal{C}_c}(c), s_n), \mathcal{S})$ where code fragment c only contains:*

- *a routine call r, together with*
- *all the code of the routines directly or indirectly called from r.*

Therefore, in a routine node, s_0 and s_n are, respectively, the states just before and after the execution of the called routine. Almost all implementations reduce c to the routine call, and they skip the code of the own routine.

Definition 8.2.13 (Routine Tree) *A* Routine Tree *is an ET where all nodes are routine nodes.*

8.2.3 Search strategies for Algorithmic Debugging

Once the ET is built, AD uses a search strategy to select one node. During many years, the main goal of most AD researchers has been the definition of better search strategies to reduce the search space after every answer, and to reduce the complexity of the questions. A survey of search strategies for AD can be found in [97]. In our formalization, a search strategy is just a function that analyzes the ET and returns an execution node (either the next node to ask or a buggy node).

Definition 8.2.14 (Search Strategy) *A* search strategy *is a function whose input is an ET $T = (N, E)$ and whose output is an execution node $n = (\mathcal{B}, \mathcal{S}) \in N$ such that:*

1. $\mathcal{S} = \text{undefined}$*, or*

2. *n is a buggy node.*

8.2.4 Algorithmic Debugging transformations

Some of the last research developments in AD have focussed on the definition of transformations of the ET. The goal of these transformations is to improve the structure of the ET before the debugging session starts, so that search strategies become more efficient. Some of these transformations cannot be applied to a routine tree. For this reason, we include this section to classify the kinds of transformations that have been defined so far, and establish a hierarchy so that future transformations can be also classified in.

There exist three essential elements in the front-end of an algorithmic debugger. The modification of any of them can lead to a different final output of the front-end (i.e., a different ET). Therefore, we classify the transformations into three different levels:

- Transformations of the *source code*: Transformations of the source code such as *inlining* are used to reduce the size of the ET by hiding routines. Contrarily, transformations such as *Loop Expansion* [56] are used to augment the size of the ET to reduce the granularity of the reported buggy code (a loop instead of a routine). In general, users should not be aware of the internal transformations applied by the debugger, thus the code fragment shown to the user should be the original code.

- Transformations of the *execution*: Transforming the way in which the source code is executed can change the generated ET. One example is changing eager evaluation by lazy evaluation. Another example is passing arguments by value instead of passing them by reference. We are not aware of any implementation that includes this kind of transformations.

- Transformations of the *ET*: Transforming the ET can significantly reduce the number of generated questions. In general, the ET is transformed with the aim of making search strategies to behave as a dichotomic search. Hence, they try to produce balanced ETs [55], or also deep trees that can be cut in the middle. Other transformations such as *Tree compression* [24] try to avoid the repetition of questions about the same routine, or try to improve the understandability of questions. This is the case of the *Node simplification* transformation, which reduces all terms to normal form [16].

Given a computation $\mathcal{C}_c$ produced by a code fragment c, being $T = (N, E)$ the ET of $\mathcal{C}_c$, we represent with $\mathcal{T}(T)$ either: (i) a transformation T' of T, (ii) the ET of a computation $\mathcal{C}'_c$ transformed from $\mathcal{C}_c$, or (iii) the ET of a computation $\mathcal{C}_{c'}$ produced by a code fragment c' transformed from c. Hence, $\mathcal{T}(T)$ is the ET produced after applying one of the previously described transformations. Note that we use T to refer to the ET that would have been generated without applying the transformation. A transformation must have some specific properties to be useful, otherwise, we run the risk of improving the structure of

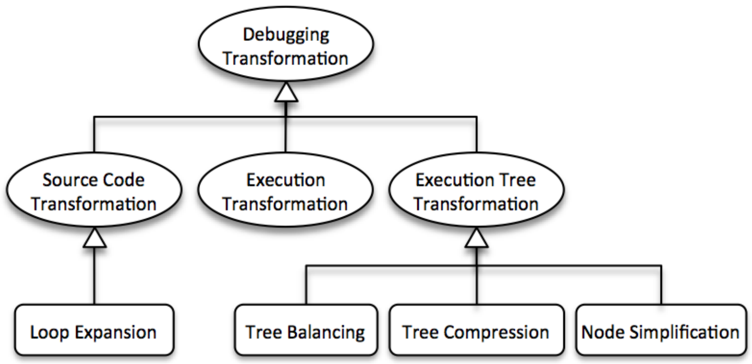


Figure 8.2: AD transformation hierarchy

the ET at the cost of losing buggy nodes. We propose four different properties to measure the impact of a transformation on a given ET: *Buggy Node Completeness*, *Buggy Code Completeness*, *Buggy Code Reduction*, and *Bug Existence*. The last one must be satisfied by any transformation to ensure that AD is still able to find at least one bug. All of them are formally presented below.

Property 1 (Buggy Node Completeness) *Given an ET $T = (N, E)$, and its transformed version $\mathcal{T}(T) = (N', E')$, $\forall n \in N$, if n is a buggy node in T then n is a buggy node in $\mathcal{T}(T)$.*

Because fulfillment of Property 1 implies that all buggy nodes belong to both ETs, then, by Definition 8.2.9, the same buggy code behaviours are detectable. Hence, all bugs that can be localized in the ET generated before applying the transformation are still detectable after applying the transformation. Many transformations cannot satisfy Property 1, but they can satisfy a relaxed version:

Property 2 (Buggy Code Completeness) *Given an ET $T = (N, E)$, and its transformed version $\mathcal{T}(T) = (N', E')$, $\forall n = ((s_0, d, s_n), wrong) \in N$, n is a buggy node in T . $\exists n' = ((s'_0, d', s'_n), wrong) \in N'$, n' is a buggy node in $\mathcal{T}(T)$ and $d = d'$.*

This property ensures that all the code that can be reported as buggy in the original ET, can be also reported as buggy in the transformed ET. However, the specific context that produced the bug is not necessarily the same in the original and in the transformed version. There exist transformations that violate the last condition ($d = d'$). However, this is not a problem when d' is a subset of d, because this means that the code reported as buggy is smaller in the transformed code (i.e., precision is increased by reducing the granularity of the located bug).

Property 3 (Buggy Code Reduction) *Given an ET $T = (N, E)$, and its transformed version $\mathcal{T}(T) = (N', E')$, $\forall n = ((s_0, d, s_n), wrong) \in N$, n is a buggy node in T . $\exists n' = ((s'_0, d', s'_n), wrong) \in N'$, n' is a buggy node in $\mathcal{T}(T)$ and $d' \subseteq d$.*

Sometimes, these properties are further relaxed to a bug existence property.

Property 4 (Bug Existence) *Given an ET $T = (N, E)$, and its transformed version $\mathcal{T}(T) = (N', E')$, If $\exists n \in N$, n is a buggy node in T, then $\exists n' \in N'$, n' is a buggy node in $\mathcal{T}(T)$.*

Property 4 is the minimum exigible requirement to ensure that the transformation has not hidden all bugs. Thus, all transformations must satisfy at least Property 4.

In Figure 8.2 we classify four AD transformations already available in the state of the art. Two of them, *Tree balancing* and *Loop expansion* produce ETs that are not routine trees. The *Tree balancing* technique satisfies Property 1, except for *collapse nodes* that only satisfy Property 2, thus, all buggy nodes (except those replaced by collapse nodes) are preserved. However buggy nodes can stop being traceable after this transformation. The *Loop expansion* technique satisfies Property 3, but not Property 2 (and thus, it does not satisfy Property 1 either). The reduction in the buggy code is obtained by introducing the ability in the ET of detecting bugs inside loops. The *Tree compression* technique satisfies Property 2, but not Property 1. The *Node simplification* technique satisfies Property 4, but not Properties 1, 2 and 3, because some buggy nodes can disappear after the transformation.

8.2.5 An Algorithmic Debugging scheme

Finally, we describe Algorithm 28, a general schema of an algorithmic debugger that includes all phases, from the generation of the ET to the reported bug. This algorithm gives an idea of how and when, the ET, the transformations, the oracle, and the search strategies participate in the whole debugging process.

Algorithm 28 Main algorithm of an Algorithmic Debugger

```
Input: A program $P$ and its input $i$.
Output: A buggy code $c$ in $P$, or $\perp$ if no bug is detected in $P$.
Initializations: $\mathcal{A} = \emptyset$          // Set of answers provided by the oracle

begin
  1) $T = getExecutionTree(P, i)$
  2) $n = debugTree(T)$
  3) return $getCode(n, T)$
end

function $debugTree(T = (N, E))$
begin
  1) while $(\exists(\mathcal{B}', \mathcal{S}') \in N, \mathcal{S}' = undef \vee wrong)$
  2)      $(\mathcal{B}, \mathcal{S}) = selectNode(T)$     // Search strategy
  3)      if $(\mathcal{S} = wrong)$ then
  4)          return $(\mathcal{B}, \mathcal{S})$
  5)      $answer = askOracle(\mathcal{B})$
  6)      $\mathcal{A} = \mathcal{A} \cup (\mathcal{B}, answer)$
  7)      $updateStates(\mathcal{A}, N)$
  8)      $T = executionTreeTransformations(T)$
  9) return $\perp$
end

function $getExecutionTree(P, i)$
begin
  1) $P' = sourceCodeTransformations(P)$
  2) $\mathcal{E}_{P'} = executeProgram(P', i)$
  3) $\mathcal{E}'_{P'} = executionTransformations(\mathcal{E}_{P'})$
  4) $T = generateExecutionTree(\mathcal{E}'_{P'})$
  5) $T' = executionTreeTransformations(T)$
  6) return $T'$
end

function $getCode(n, T = (N, E))$
begin
  1) if $(n = ((s_0, d, s_n), \mathcal{S}))$ then
  2)      return $d \setminus \bigcup_{(n,((s'_0,d_i,s'_n),\mathcal{S}'))\in E} d_i$
  3) return $\perp$
end
```

The main function performs the two phases of AD (Lines 1-2) and then returns a buggy code of the program (Line 3). In the first phase (*getExecutionTree* function) the ET is created performing all possible transformations in the source code (Line 1), in the execution (Line 3) and in the ET (Line 5). Once the ET is created, the second phase (*debugTree* function) starts. During this phase, the debugger traverses the ET selecting nodes with a search strategy (Line 2). The *selectNode* function is an implementation of one of the search strategies in the literature. There have been a lot of research for more than a decade concerning which should be the node to ask about. A survey can be found in [97]. No matter which search strategy is used, *selectNode* returns a node to ask about (the state of the node is *undefined*), or a buggy node (the state of the node is *wrong* (Line 3)). Once a node has been selected, the debugger asks the oracle about its correctness (Line 5). The oracle provides the intended interpretation to the algorithm. With the oracle's answer, the debugger updates the state of the nodes of the ET (Lines 5-7). Note that the oracle's answer can affect the state of several nodes. This effectively changes the information of the ET, and thus, at this point, a new ET transformation could be used to optimize the ET (Line 8). Then, the process is repeated selecting more nodes. When the search strategy finds a buggy node (Lines 3-4) or it cannot select more nodes (Line 1) the second phase finishes and the debugger returns (see *getCode* function) the buggy code associated with the located buggy node (see Lemma 8.2.11), or it returns a message indicating that there does not exist a bug (it is indicated with $\perp$ in Line 3), respectively. The last case happens, e.g., when all nodes are reported as *correct*.

8.3 Related work

AD has been applied to all mature languages. All current implementations use a sort of ET to represent computations. Even in those lazy implementations of AD where the execution of the front-end and the back-end is interleaved (see, e.g., [79]), the construction of the ET is needed before the program can be debugged. Along the years, each paradigm has adopted a well-defined and studied data structure to represent the ET.

8.3.1 A little bit of history

AD started in the seminal work by Shapiro with the notion of contradiction backtracking using "crucial experiments" within Popper's philosophical dictum of conjectures and refutations [91]. Hence, the first notion of ET appeared in the context of the logic paradigm. Shapiro used refutation trees as ETs. Later implementations of AD in the logic paradigm such as NU-Prolog [102, 4] also used refutation trees.

In the context of the functional paradigm, the ET data structure was proposed by Henrik Nilsson and Jan Sparud: The Evaluation Dependence Tree (EDT). They first proposed this data structure as a record of the execution [81], and then, as an appropriate ET for AD [82]. The EDT is particularly useful to represent lazy computations by hiding non-computed terms. In fact, the own EDT can be computed lazily as in [79]. The most successful implementations of AD for the functional paradigm are based on the EDT. Notable cases are the Freja [78], Hat-Delta [23], and Buddha [85] debuggers.

In multi-paradigm languages such as Mercury, TOY, or Curry, the ET is also represented with either a proof tree or an EDT. Examples of debuggers for these languages are the Mercury Debugger [68], the Münster Curry Debugger [66], DDT [10], and B.i.O [8].

In the imperative paradigm, a redefinition of the EDT was used. It has been often called *Execution Tree* [31, 46], but, conceptually, it is equivalent to the EDT, and it can be seen as a dynamic version of the *Call Graph* where every single call generates a different node in the graph, and thus no cycles are possible (i.e., it is a tree).

8.3.2 Modern implementations

All the debuggers mentioned in the previous sections are somehow "standard" in the sense that they are based on the standard definition of the ET (either the refutation trees or the EDT). However, in the last 5 years, there has been a new trend in AD tools: Researchers have implemented new techniques that go beyond the standard definition of the ET. Contrarily to the previously described tools, modern algorithmic debuggers are not standalone tools. They are plugins that can be integrated as part of an IDE. Examples of these debuggers are JHyde [43] and HDJ [37], being both of them part of Eclipse. Precisely because they are integrated into a development environment, they have direct access to dynamic information—they can even manipulate the JVM at runtime—that can be used to enhance the debugging sessions. In particular, the following techniques go beyond the standard ET: (i) *Tree compression* hides nodes of the ET (it breaks the standard parent-child relation in the ET). (ii) *Tree balancing* introduces new artificial nodes in the ET (it breaks the standard definition of ET node). (iii) *Loop expansion* and (iv) *ET zooming* decompose ET nodes (they break the standard definition of ET node). We are not aware of any definition of ET able to represent the previous four techniques.

Part IV

Conclusions and Future Work

Chapter 9

Conclusions

Software debugging includes the detection and the correction of syntactical as well as semantical errors in the source code. There have been important advances in the debugging of syntactical errors and, nowadays, the debugging of syntactical errors is one of the most advanced and automated software development processes, tackled with the use of modern compilers that use grammars to completely specify a programming language, and analyses such as LL(n), LALR, and LR(n). The situation is very different with respect to the detection and correction of semantical errors (those that make the system crash, not to do what it is expected, or do it in an inefficient way). In order to guaranty the software quality, it is essential that the software production system incorporates effective and efficient mechanisms for the debugging of semantical errors.

Given the complexity and the size of current software systems, debugging can become an arduous task with an unpredictable difficulty and duration. However, surprisingly, debugging is one of the software processes that has been less treated by the scientific community. In fact, the same debugging techniques that were used twenty years ago, are still used nowadays [96]. The main problem of current techniques is that they help the programmer to find information about the debugged program, but they do not further use that information to help the programmer to guide the search for the own error. This is the case, for example, of the breakpoints-based techniques.

There exist, however, several automatic or semi-automatic debugging techniques that guide the search for the bug and, under some conditions, they are able to guaranty that they will eventually find a bug in the code. Among these techniques, we can highlight Program Slicing [103] initially introduced in the imperative paradigm, and Algorithmic Debugging (AD) [92] initially introduced in the declarative paradigm.

AD was created 34 years ago, and many efforts have been made to enhance this technique. For instance, the technique has been adapted to the logic, functional and imperative paradigms, as well as several algorithmic debuggers have been created for each of them [75, 78, 23, 10, 68, 41, 85, 66, 46]. Moreover, there have emerged at least 17 different AD search strategies [95] and, in general, each of them was proposed in the context of a different language (for instance, Top-Down [92] for Prolog, Subterm Dependency Tracking [68] for Mercury, Hat-Delta [24] for Haskel, Divide by YES [95] and JHyde's strategy [43] for Java, etc.) and frequently their adaptation to other languages is not straightforward. In addition, many techniques have been developed to increase the usability of the technique. This is the case for example of: Tree Compression [24], which removes the nodes of the tree that are redundant; the *I don't know* and *Inadmissible* answers [84], which allows for directing the search for the bug in those situations in which the original answers are useless; the Mercury debugger [68], which uses subexpression information to enhance the search for the bug; and the JHyde debugger [43], which uses the own Eclipse's views to show the AD information to the user.

Even so, AD still has several drawbacks that need to be achieved. One of the main problems of the AD technique is its scalability. When the system to debug performs big computations, the data structures that are generated to store those computations may occupy gigabytes, which makes most current debuggers not support big programs. This is the case, for instance, of the DDT debugger [10], the Buddha debugger [85] of Haskel, and the Curry language debugger, among others. An approximation that has been carried out is to store these data structures that store the computation in secondary memory (e.g., the redex trail of the Hat debugger of the Haskell language is stored in several files). However, this solution solves the memory consumption problem at the cost of an answer time reduction, which is unacceptable in many cases. In fact, the time needed to start the debugging session is still far from being practical. The user has to wait until the whole data structure is stored in main (or secondary) memory to start the debugging session. In addition, the amount of unnecessary questions generated slow down the debugging session, and when the debugging session is finally concluded, the algorithmic debugger only provides the method that contains the bug, forcing the user to start a new debugging session with another debugger to find the bug inside this method.

All these problems have been tackled during the development of this thesis. In particular, there have been advances with respect to the adaptation of the current AD techniques to other languages. Concretely, it has been developed the first Declarative Debugger for Java (DDJ) (see Section 7.1) that (1) incorporates 13 different search strategies, allowing for alternating between these strategies during the same debugging sessions; and (2) incorporates four different techniques, allowing for reducing the amount of questions performed by the debugger. The adaptation of AD techniques between different paradigms opens a new and promising research field that consists in the combination of AD techniques that have never coexisted in the same language. To complement DDJ, the Hybrid Debugger for Java (HDJ) (see Section 7.2) has been developed, which allows the user to use either DDJ, a *Trace Debugger* or an *Omniscient Debugger* depending on which type of debugger better helps the user at that point.

In this thesis we have solved some of the drawbacks of AD with the aim of producing a practical and usable debugger. The main contributions of this thesis have been summarized below:

1. **Scalability.**
   Since AD was proposed, a debugging session has been traditionally divided into two independent and sequential phases: obtaining the execution tree, and debugging the program. During the development of the DDJ debugger, we developed the *Virtual Execution Trees* technique (see Section 5.1), which allows the debugger to overlap these two phases, permitting for reducing the time that a user needs to start the debugging session from minutes to milliseconds. Moreover, the scalability problem has been encompassed incorporating to the debugger a cache-based architecture. This new architecture allows the debugger to be scalable in time and in memory. Moreover, in the HDJ debugger, three different debuggers have been combined to increase the scalability, the first of which is the own *Trace Debugger* of Eclipse. Its combination with DDJ improves the whole debugging sessions. By using the *Trace Debugger*, the user can place breakpoints into the source code to direct the search until a piece of code that contains the bug is found. Later, AD can be used to debug this piece of code in an abstract way, which helps to debug complex algorithms because it is not needed to understand how they are implemented, but only what they are supposed to do. This combination reduces the amount of information that AD has to store, because now, the debugger only stores the part of the code in which the user thinks the error is. Thus, the debugger only needs to store a suspicious subcomputation instead of the whole computation.

2. **Usability.**
   The data structure used to store the computation of the debugged program (the execution tree) is highly related with the usability of the technique. The more unbalanced it is, the more time is

needed to explore it and hence to find the bug. When it is completely unbalanced, the user may be asked a linear number of questions with respect of the number of nodes of the tree, whereas when the tree is completely balanced, the debugger may only ask a logarithmic number of questions. Clearly, the structure of the tree is crucial for the technique. To improve the structure of the ET, we have developed techniques that are now integrated into DDJ and we have also developed some strategies that reduce the number of questions. In particular, DDJ implements the *Tree Balancing* technique (see Section 5.2) that balances the tree by combining some nodes, and the *Tree Compression* technique (see Section 5.5) that removes some nodes of the tree when this change balances it. Moreover, *Optimal Divide & Query* (see Section 6.2) improves the best known search strategy (D&Q), and hence it reduces the number of questions that the user is asked about. Finally, the HDJ plugin for Eclipse has been developed allowing the user to combine DDJ with its *Trace Debugger*, and to use DDJ in a more familiar context.

3. **Granularity.**
   AD has the drawback of being only able to find which is the method that contains the bug, instead of the expression. DDJ includes now a novel technique called *Loop Expansion* (see Section 5.4) that increases the granularity by converting the loops presented in the source code into recursive calls. This transformation produces a completely different data structure in which the own loops and theirs iterations are represented, helping the user to also discard loops of causing the bug. In addition, when DDJ has found the portion of code that contains the bug, the *Omniscient Debugger* included in HDJ can be used to further explore it. This debugger complements DDJ by permitting the user to find the expression that contains the bug.

4. **Uniformity.**
   The original formulation of AD was oriented to the logic paradigm, but it was quickly adapted to the functional and imperative paradigm respecting the basis of the formulation. However, this formulation is obsolete nowadays, and frequently researchers need to redefine specific parts of the formulation. In this thesis we have included a reformulation of the technique (see Chapter 8) that is paradigm-independent, and that can be used as a formal representation of all current existing techniques at the time of writing these lines. Moreover, it provides a classification of transformation techniques that affect the data structure of AD, as well as the properties that these transformations may fulfil. Researchers can use this classification to easily classify their AD techniques. The classification of a technique is useful because it provides other researchers with information about the properties that holds for this technique. In addition, an AD scheme is presented to provide a general view of how all the components of AD interact, helping the interested reader to better understand the technique.

As a result of this thesis, HDJ is the first algorithmic debugger that incorporates almost all search strategies from different paradigms, that overlaps both AD phases, that incorporates new AD techniques such as *Virtual Execution Tree*, *Tree Balancing*, *Loop Expansion*, and *Tree Compression*, and that combines different debuggers that share their information and collaborate in a single debugging session. In summary, HDJ can be considered as the most advanced algorithmic debugger that exists either in the logic, functional or imperative paradigm (see [18]).

Chapter 10

Open Lines of Research

There exist several aspects in which Algorithmic Debugging can be improved. We list some of them in the following points:

Techniques The *Balancing* technique shown in Section 5.2 is oriented to produce ETs that can be quicker debugged using Top-Down strategies. ETs that are more suitable for *Divide and Query* strategies are more difficult to produce due to the internal nature of the own search strategies. They can "jump" from one node to another without any observable relationship. The nodes are introduced by the technique because of one main objective: the introduced nodes will be more likely to be selected by the strategy than other nodes. In presence of *Divide and Query* strategies this objective is more difficult to achieve. During the first steps of the balancing technique, the projection nodes can be easily added, but as the process requires to create more projection nodes, the next introduced nodes can prevent the search strategy to select the first introduced ones. A mechanism to properly produce balanced ETs to be used with *Divide and Query* strategies is needed. With respect to the *Tree Compression* technique shown in Section 5.5, it produces ETs that are better than the input ones. However, in order to produce a tree compression in a reasonable time, the technique tries to find a local minimum. Obtaining a global minimum would compress the ET in such a way that the debugger would ask less questions to the user.

Strategies Despite the *Optimal Divide and Query* strategy shown in Section 6.2 has been defined and implemented into *DDJ*, the *Divide by Queries* strategy shown in Section 6.4 is not present. The reason is that *Optimal Divide and Query* can be implemented with a linear cost whereas the problem that we want to solve in the *Divide by Queries* strategy is a NP-hard problem as demonstrated in [57]. In fact, in our experiments we need around half an hour to obtain which is the optimal node in an ET with 20 nodes using a brute-force algorithm. In an initial research, we have produced three properties that an optimal node must fulfill, and these properties have allow us to obtain the same optimal nodes as the obtained by the brute-force algorithm but only in a few seconds at the most. However, despite the computation time needed to solve the problem has been considerably decreased, a scalable approximation able to obtain the optimal node in bigger ETs is needed, or at least to obtain a set composed of a few nodes, one of which being the optimal node.

Implementations *DDJ* has been evolved into *HDJ*, a debugger that mix DDJ with *Trace Debugging* and *Omniscient Debugging*. With respect to the *Trace Debugger*, it is the default debugger of Eclipse, hence it implements a lot of features that helps the user to find the bug. In contrast, the

Omniscient Debugger of *HDJ* has been implemented from scratch, and only the most important features of the debugger are present. In order to provide the user with a complete and perfect rapport among all debuggers, the *Omniscient Debugger* implemented in *HDJ* must be improved to provide at least all the features present in its current state of the art.

Bibliography

[1] *ActiveState Komodo*. Available at URL: `http://komodoide.com/`. 2000.

[2] Evyatar Av-Ron. "Top-Down Diagnosis of Prolog Programs". PhD thesis. Weizmanm Institute, 1984.

[3] Henry Givens Baker. *Garbage Collection, Tail Recursion and First-Class Continuations in Stack-Oriented Languages*. Patent. US 5590332. 1996. URL: `http://www.patentlens.net/patentlens/patent/US_5590332/en/`.

[4] Tim Barbour and Lee Naish. *Declarative Debugging of a Logical-Functional Language*. Tech. rep. University of Melbourne, 1994.

[5] Dominic Binks. "Declarative Debugging in Gödel". PhD thesis. University of Bristol, 1995.

[6] *Borland JBuilder*. Available at URL: `http://www.embarcadero.com/products/jbuilder/`. 2008.

[7] Franois Bourdoncle. "Abstract Debugging of Higher-Order Imperative Languages". In: *ACM SIGPLAN Notices* 28.6 (1993), pp. 46–55. ISSN: 0362-1340.

[8] Bernd Braßel and Holger Siegel. "Debugging Lazy Functional Programs by Asking the Oracle". In: *Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages*. Ed. by Olaf Chitil, Zoltán Horváth, and Viktória Zsók. Vol. 5083. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 183–200. ISBN: 978-3-540-85372-5.

[9] J. Mark Bull, L. A. Smith, Martin D. Westhead, D. S. Henty, and R. A. Davey. "A Benchmark Suite for High Performance Java". In: *Concurrency: Practice and Experience* 12.6 (2000), pp. 375–388. ISSN: 1096-9128. DOI: `10.1002/1096-9128(200005)12:6<375::AID-CPE480>3.0.CO;2-M`. URL: `http://dx.doi.org/10.1002/1096-9128(200005)12:6<375::AID-CPE480>3.0.CO;2-M`.

[10] Rafael Caballero. "A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs". In: *Proceedings of the 2005 ACM-SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP'05)*. Tallinn, Estonia: ACM Press, 2005, pp. 8–13. ISBN: 1-59593-069-8. DOI: `http://doi.acm.org/10.1145/1085099.1085102`.

[11] Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez. "Declarative Debugging of Wrong and Missing Answers for SQL Views". In: *Proceedings of the 11th International Symposium on Functional and Logic Programming, FLOPS 2012*. Ed. by Tom Schrijvers and Peter Thiemann. Vol. 7294. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 73–87. ISBN: 978-3-642-29822-6.

[12] Rafael Caballero, Christian Hermanns, and Herbert Kuchen. "Algorithmic Debugging of Java Programs". In: *Proceedings of the 2006 Workshop on Functional Logic Programming (WFLP'06)*. Madrid, Spain: Electronic Notes in Theoretical Computer Science, 2006, pp. 63–76.

[13] Rafael Caballero, Narciso Martí-Oliet, Adrián Riesco, and Alberto Verdejo. "A Declarative Debugger for Maude Functional Modules". In: *Electronic Notes Theoretical Computer Science* 238.3 (3 2009), pp. 63–81. ISSN: 1571-0661. DOI: `http://dx.doi.org/10.1016/j.entcs.2009.05.013`.

[14] Rafael Caballero, Enrique Martin-Martin, Adrián Riesco, and Salvador Tamarit. “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.

[15] Rafael Caballero, Enrique Martin?Martin, Adrián Riesco, and Salvador Tamarit. “A Declarative Debugger for Concurrent Erlang Programs”. In: *Proceedings of the 15th Spanish Workshop on Programming Languages (PROLE 15)*. Universidad de Cantabria, 2015.

[16] Rafael Caballero, Adrián Riesco, Alberto Verdejo, and Narciso Martí-Oliet. “Simplifying Questions in Maude Declarative Debugger by Transforming Proof Trees”. In: *21st International Symposium Logic-Based Program Synthesis and Transformation (LOPSTR 2011)*. Ed. by Germán Vidal. Vol. 7225. Lecture Notes in Computer Science. Springer, 2011, pp. 73–89. ISBN: 978-3-642-32210-5.

[17] Miguel Calejo. “A Framework for Declarative Prolog Debugging”. PhD thesis. New University of Lisbon, 1992.

[18] Diego Cheda and Josep Silva. “A Comparative of Algorithmic Debuggers”. In: *Proceedings of the 7th Spanish Workshop on Programming Languages (PROLE 07)*. 2007, pp. 171–180. ISBN: 978-84-9732-599-8.

[19] Diego Cheda and Josep Silva. “State of the Practice in Algorithmic Debugging”. In: *Electronic Notes in Theoretical Computer Science* 246 (2009), pp. 55–70.

[20] William Clinger. “Proper Tail Recursion and Space Efficiency”. In: *ACM SIGPLAN Notices* 33.5 (1998), pp. 174–185. ISSN: 0362-1340. DOI: `10.1145/277652.277719`. URL: `http://doi.acm.org/10.1145/277652.277719`.

[21] Marco Comini, Giorgio Levi, Maria Chiara Meo, and Giuliana Vitiello. “Abstract Diagnosis”. In: *The Journal of Logic Programming* 39.1-3 (1999), pp. 43–93. ISSN: 0743-1066. DOI: `http://dx.doi.org/10.1016/S0743-1066(98)10033-X`. URL: `http://www.sciencedirect.com/science/article/pii/S074310669810033X`.

[22] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL’77)*. 1977, pp. 238–252.

[23] Thomas Davie and Olaf Chitil. “Hat-delta: One Right Does Make a Wrong”. In: *Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages (IFL’05)*. Ed. by Andrew Butterfield. 2005, p. 11.

[24] Thomas Davie and Olaf Chitil. “Hat-delta: One Right Does Make a Wrong”. In: *Proceedings of the 7th Symposium on Trends in Functional Programming (TFP’06)*. 2006.

[25] *Eclipse*. Available at URL: `http://www.eclipse.org/`. 2003.

[26] Erik Elmroth and Fred Gehrung Gustavson. “Applying Recursion to Serial and Parallel QR Factorization Leads to Better Performance”. In: *IBM Journal of Research and Development* 44.4 (2000), pp. 605–624. DOI: `10.1147/rd.444.0605`. URL: `http://dx.doi.org/10.1147/rd.444.0605`.

[27] Madalina Erascu and Tudor Jebelean. “A Purely Logical Approach to the Termination of Imperative Loops”. In: *13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* 0 (2010), pp. 142–149. DOI: `http://doi.ieeecomputersociety.org/10.1109/SYNASC.2010.64`.

[28] Maarten Faddegon and Olaf Chitil. “Algorithmic Debugging of Real-World Haskell Programs: Deriving Dependencies From the Cost Centre Stack”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. Vol. 50. 2015, pp. 33–42.

[29] Andrzej Filinski. “Recursion From Iteration”. In: *LISP and Symbolic Computation* 7 (1 1994), pp. 11–37. ISSN: 0892-4635. URL: `http://dx.doi.org/10.1007/BF01019943`.

[30] Alcides Fonseca. *AEminium/loops2recursion Java Library*. Available at URL: `https://github.com/AEminium/loops2recursion/`. 2012.

[31] Peter Fritzson, Nahid Shahmehri, Mariam Kamkar, and Tibor Gyimóthy. “Generalized Algorithmic Debugging and Testing”. In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.4 (1992), pp. 303–322.

[32] Andy Georges, Dries Buytaert, and Lieven Eeckhout. “Statistically Rigorous Java Performance Evaluation”. In: *SIGPLAN Not.* 42.10 (2007), pp. 57–76. ISSN: 0362-1340. DOI: `10.1145/1297105.1297033`. URL: `http://doi.acm.org/10.1145/1297105.1297033`.

[33] Paul Gestwicki and Bharat Jayaraman. “JIVE: Java Interactive Visualization Environment”. In: *Companion to the 19th Annual ACM-SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’04)*. Vancouver, BC, CANADA: ACM Press, 2004, pp. 226–228. ISBN: 1-58113-833-4.

[34] Herman Geuvers. “Inductive and Coinductive Types with Iteration and Recursion”. In: *Proceedings of the Workshop on Types for Proofs and Programs*. 1992, pp. 193–217.

[35] David Giammona. *ORACLE ADF - Putting It Together*. Tech. rep. ADF Declarative Debugger Archives, 2009. URL: `http://blogs.oracle.com/DavidGiammona/adf_declarative_debugger`.

[36] Hani Girgis and Bharat Jayaraman. *JavaDD: a Declarative Debugger for Java*. Tech. rep. University at Buffalo, 2006.

[37] Juan González, David Insa, and Josep Silva. “A New Hybrid Debugging Architecture for Eclipse”. In: *Proceedings of the 23rd International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2013)*. Vol. 8901. Lecture Notes in Computer Science (LNCS). Springer, 2014, pp. 183–201.

[38] Francisco González-Blanch, Reyes De Miguel, and Susana Serrano. “Depurador Declarativo de Programas Java”. Available at URL: `http://eprints.ucm.es/9114/`. MA thesis. Universidad Complutense de Madrid, 2006.

[39] Fred Gehrung Gustavson. “Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms”. In: *IBM Journal of Research and Development* 41.6 (1997), pp. 737–756. DOI: `10.1147/rd.416.0737`. URL: `http://dx.doi.org/10.1147/rd.416.0737`.

[40] Chris Hanson. “Efficient Stack Allocation for Tail-Recursive Languages”. In: *Proceedings of the 1990 ACM conference on LISP and Functional Programming (LFP’90)*. Nice, France: ACM, 1990, pp. 106–118. ISBN: 0-89791-368-X. DOI: `10.1145/91556.91603`. URL: `http://doi.acm.org/10.1145/91556.91603`.

[41] Michael Hanus. *Curry: An Integrated Functional Logic Language (Version 0.8.2 of March 28, 2006)*. Available at URL: `http://www.informatik.uni-kiel.de/~curry/`. 2006.

[42] Peter George Harrison and Hessam Khoshnevisan. “A New Approach to Recursion Removal”. In: *Electronic Notes in Theoretical Computer Science* 93.1 (1992), pp. 91–113. ISSN: 0304-3975. DOI: `10.1016/0304-3975(92)90213-Y`. URL: `http://dx.doi.org/10.1016/0304-3975(92)90213-Y`.

[43] Christian Hermanns and Herbert Kuchen. “Hybrid Debugging of Java Programs”. In: *Software and Data Technologies*. Ed. by María José Escalona, José Cordeiro, and Boris Shishkov. Vol. 303. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2013, pp. 91–107. ISBN: 978-3-642-36176-0. DOI: `10.1007/978-3-642-36177-7_6`. URL: `http://dx.doi.org/10.1007/978-3-642-36177-7_6`.

[44] Visit Hirunkitti and Christopher John Hogger. “A Generalised Query Minimisation for Program Debugging”. In: *Proceedings of the 1st International Workshop of Automated and Algorithmic Debugging (AADEBUG’93)*. Springer LNCS 749, 1993, pp. 153–170.

[45] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. “Design and Implementation of a Backward-In-Time Debugger”. In: *Proceedings of NODe 2006*. Vol. P-88. Erfurt, Germany: Lecture Notes in Informatics, 2006, pp. 17–32.

[46] David Insa and Josep Silva. “An Algorithmic Debugger for Java”. In: *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010)*. 2010, pp. 1–6. ISBN: 978-1-4244-8630-4.

[47] David Insa and Josep Silva. “Scaling Up Algorithmic Debugging with Virtual Execution Trees”. In: *Proceedings of the 20th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2010)*. Vol. 6564. Lecture Notes in Computer Science. Springer, 2010, pp. 149–163.

[48] David Insa and Josep Silva. “An Optimal Strategy for Algorithmic Debugging”. In: *Proceedings of the 26th International Conference on Automated Software Engineering (ASE 2011)*. Ed. by Perry Alexander, Corina S. Pasareanu, and John G. Hosking. IEEE Computer Society, 2011, pp. 203–212. ISBN: 978-1-4577-1638-6.

[49] David Insa and Josep Silva. “Hacia una estrategia óptima para la Depuración Algorítmica”. In: *Proceedings of the 11th Spanish Workshop on Programming Languages (PROLE 11)*. 2011.

[50] David Insa and Josep Silva. “Optimal Divide and Query (extended version)”. In: *Computing Research Repository (`http://arxiv.org/abs/1107.0350`)* (2011).

[51] David Insa and Josep Silva. “Implementation of an Optimal Strategy for Algorithmic Debugging”. In: *Electronic Notes in Theoretical Computer Science* 282 (2012), pp. 47–60.

[52] David Insa and Josep Silva. *loops2recursion Java Library*. Available at URL: `http://www.dsic.upv.es/~jsilva/loops2recursion/`. 2013.

[53] David Insa and Josep Silva. “A Generalized Model for Algorithmic Debugging”. In: *Proceedings of the 25th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2015)*. Ed. by Moreno Falaschi. Vol. 9527. Lecture Notes in Computer Science (LNCS). Springer, 2015, pp. 261–276.

[54] David Insa and Josep Silva. “Automatic Transformation of Iterative Loops into Recursive Methods”. In: *Information and Software Technology* 58 (2015), pp. 95–109.

[55] David Insa, Josep Silva, and Adrián Riesco. “Speeding up Algorithmic Debugging using Balanced Execution Trees”. In: *Proceedings of the 7th International Conference Tests and Proofs (TAP 2013)*. Ed. by Margus Veanes and Luca Viganò. Vol. 7942. Lecture Notes in Computer Science (LNCS). Springer, 2013, pp. 133–151. ISBN: 978-3-642-38915-3.

[56] David Insa, Josep Silva, and César Tomás. “Enhancing Declarative Debugging with Loop Expansion and Tree Compression”. In: *Proceedings of the 22nd International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2012)*. Ed. by Elvira Albert. Vol. 7844. Lecture Notes in Computer Science (LNCS). Springer, 2012, pp. 71–88. ISBN: 978-3-642-38196-6, 978-3-642-38197-3.

[57] Tobias Jacobs, Ferdinando Cicalese, Eduardo Sany Laber, and Marco Molinaro. "On the Complexity of Searching in Trees: Average-Case Minimization". In: *Proceedings of the 36th International Colloquium on Automata, Languages and Programming (ICALP)*. Ed. by Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis. Vol. 6198. Lecture Notes in Computer Science (LNCS). Springer, 2010, pp. 527–539.

[58] Gabriella Kókai, Jörg Nilson, and Christian Niss. "GIDTS: A Graphical Programming Environment for Prolog". In: *Proceedings of the 2nd Workshop on Program Analysis for Software Tools and Engineering (PASTE'99)*. Toulouse, France: ACM Press, 1999, pp. 95–104.

[59] Hoon-Joon Kouh and Weon-Hee Yoo. "The Efficient Debugging System for Locating Logical Errors in Java Programs". In: *Proceedings of the 2003 Computational Science and Its Applications (ICCSA 2003)*. Ed. by Vipin Kumar, Martina L. Gavrilova, Chih Jeng Kenneth Tan, and Pierre L'Ecuyer. Vol. 2667. Lecture Notes in Computer Science. Montreal, Canada: Springer, 2003, pp. 684–693. ISBN: 3-540-40155-5.

[60] Bil Lewis. "Debugging Backwards in Time". In: *Available in the Computing Research Repository (http://arxiv.org/abs/cs.SE/0310016)* cs.SE/0310016 (2003).

[61] Adrian Lienhard, Tudor Girba, and Oscar Nierstrasz. "Practical Object-Oriented Back-in-Time Debugging". In: *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*. Springer, 2008, pp. 592–615.

[62] Yanhong A. Liu and Scott D. Stoller. "From Recursion to Iteration: What Are the Optimizations?" In: *Proceedings of the 2000 ACM-SIGPLAN Workshop on Partial Evaluation and semantics-based Program Manipulation (PEPM'00)*. Boston, Massachusetts, United States: ACM, 2000, pp. 73–82. ISBN: 1-58113-201-8. DOI: 10.1145/328690.328700. URL: http://doi.acm.org/10.1145/328690.328700.

[63] John Lloyd. "Declarative Error Diagnosis". In: *New Generation Computing* 5.2 (1987), pp. 133–154. ISSN: 0288-3635.

[64] Yong Luo and Olaf Chitil. "Proving the Correctness of Algorithmic Debugging for Functional Programs". In: *7th Symposium on Trends in Functional Programming (TFP 2006)*. Vol. 7. Intellect, 2006, pp. 19–34.

[65] Yong Luo and Olaf Chitil. *Algorithmic Debugging and Trusted Functions*. Tech. rep. 10-07. UK: University of Kent, 2007. URL: http://www.cs.kent.ac.uk/pubs/2007/2642.

[66] Wolfgang Lux. *Münster Curry User's Guide*. Available at URL: http://danae.uni-muenster.de/~lux/curry/user.pdf. 2006.

[67] Wolfgang Lux. "Declarative Debugging Meets the World". In: *Electronic Notes in Theoretical Computer Science* 216 (2008), pp. 65–77. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.06.034. URL: http://www.sciencedirect.com/science/article/B75H1-4SY6BJ7-5/2/34f83f409e13b1ba40307e99b4d6567b.

[68] Ian Douglas MacLarty. "Practical Declarative Debugging of Mercury Programs". PhD thesis. University of Melbourne, 2005.

[69] Machi Maeji and Tadashi Kanamori. *Top-Down Zooming Diagnosis of Logic Programs*. Tech. rep. TR-290. ICOT, Japan, 1987.

[70] Panagiotis Manolios and J Strother Moore. "Partial Functions in ACL2". In: *Journal of Automated Reasoning* 31.2 (2003), pp. 107–127. ISSN: 0168-7433. DOI: 10.1023/B:JARS.0000009505.07087.34. URL: http://dx.doi.org/10.1023/B:JARS.0000009505.07087.34.

[71] John McCarthy. “Towards a Mathematical Science of Computation”. In: *Proceedings of the 2nd International Federation for Information Processing Congress (IFIP’62)*. North-Holland, 1962, pp. 21–28.

[72] Sun Microsystems. *Java Platform Debugger Architecture - JPDA*. Available at URL: `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`. 2010.

[73] Salman Mirghasemi, John Barton, and Claude Petitpierre. *Debugging by lastChange*. Tech. rep. `http://people.epfl.ch/salman.mirghasemi`, 2011.

[74] Magnus Myreen and Michael Gordon. “Transforming Programs into Recursive Functions”. In: *Electronic Notes in Theoretical Computer Science* 240 (2009), pp. 185–200. ISSN: 1571-0661. DOI: `10.1016/j.entcs.2009.05.052`. URL: `http://dx.doi.org/10.1016/j.entcs.2009.05.052`.

[75] Lee Naish, Philip Wayne Dart, and Justin Zobel. “The NU-Prolog Debugging Environment”. In: *Proceedings of the 6th International Conference on Logic Programming (ICLP’89)*. Ed. by Antonio Porto. Lisboa, Portugal, 1989, pp. 521–536.

[76] *NetBeans*. Available at URL: `http://www.netbeans.org/`. 1999.

[77] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. New York, NY, USA: John Wiley & Sons, Inc., 1992. ISBN: 0-471-92980-8.

[78] Henrik Nilsson. “Declarative Debugging for Lazy Functional Languages”. PhD thesis. Linköping, Sweden, 1998.

[79] Henrik Nilsson. “How to Look Busy While Being as Lazy as Ever: the Implementation of a Lazy Functional Debugger”. In: *Journal of Functional Programming* 11.6 (2001), pp. 629–671.

[80] Henrik Nilsson and Peter Fritzson. “Algorithmic Debugging for Lazy Functional Languages”. In: *Journal of Functional Programming* 4.3 (1994), pp. 337–370.

[81] Henrik Nilsson and Jan Sparud. *The Evaluation Dependence Tree: An Execution Record for Lazy Functional Debugging*. Tech. rep. Department of Computer and Information Science, Linköping, 1996.

[82] Henrik Nilsson and Jan Sparud. “The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging”. In: *Automated Software Engineering* 4.2 (1997), pp. 121–150.

[83] *Omnicore CodeGuide*. Available at URL: `http://www.omnicore.com/en/codeguide.htm`. 2007.

[84] Lus Moniz Pereira. “Rational Debugging in Logic Programming”. In: *Proceedings of the Third International Conference on Logic Programming*. New York, NY, USA: Springer-Verlag LNCS 225, 1986, pp. 203–210. ISBN: 0-387-16492-8.

[85] Bernie Pope. “A Declarative Debugger for Haskell”. PhD thesis. Australia: The University of Melbourne, 2006.

[86] Guillaume Pothier. “Towards Practical Omniscient Debugging”. PhD thesis. University of Chile, 2011.

[87] Adrián Riesco. “Declarative Debugging and Heterogeneous Verification in Maude”. PhD thesis. Universidad Complutense de Madrid, 2011.

[88] Adrián Riesco, Alberto Verdejo, and Narciso Martí-Oliet. “A Complete Declarative Debugger for Maude”. In: *Proceedings of the 13th International Conference Algebraic Methodology and Software Technology (AMAST 2010)*. Vol. 6486. Lecture Notes in Computer Science. Springer-Verlag, 2010, pp. 216–225.

[89] Adrián Riesco, Alberto Verdejo, and Narciso Martí-Oliet. “Declarative Debugging of Missing Answers for Maude”. In: *Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA 2010)*. 2010, pp. 277–294.

[90] Adrián Riesco, Alberto Verdejo, Narciso Martí-Oliet, and Rafael Caballero. “Declarative Debugging of Rewriting Logic Specifications”. In: *Journal of Logic and Algebraic Programming (JLAP)* (2011).

[91] Ehud Y. Shapiro. *Inductive Inference of Theories from Facts*. Tech. rep. RR 192. Yale University (New Haven, CT US), 1981. URL: `http://opac.inria.fr/record=b1007525`.

[92] Ehud Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982.

[93] *SICStus Prolog SPIDER IDE*. Available at URL: `https://sicstus.sics.se/spider/`. 2009.

[94] Josep Silva. “Three New Algorithmic Debugging Strategies”. In: *Proceedings of the 6th Jornadas de Programación y Lenguajes (PROLE’06)*. 2006, pp. 243–252.

[95] Josep Silva. “A Comparative Study of Algorithmic Debugging Strategies”. In: *Proceedings of the 16th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR’06)*. Springer LNCS 4407, 2007, pp. 143–159.

[96] Josep Silva. “Debugging Techniques for Declarative Languages: Profiling, Program Slicing, and Algorithmic Debugging”. PhD thesis. Universidad Politécnica de Valencia, 2007.

[97] Josep Silva. “A Survey on Algorithmic Debugging Strategies”. In: *Advances in Engineering Software* 42.11 (2011), pp. 976–991. ISSN: 0965-9978. DOI: `10.1016/j.advengsoft.2011.05.024`. URL: `http://dx.doi.org/10.1016/j.advengsoft.2011.05.024`.

[98] Josep Silva. “A Vocabulary of Program Slicing-Based Techniques”. In: *ACM Computing Surveys* 44.3 (2012).

[99] Xinox software. *JCreator*. Available at URL: `http://www.jcreator.com/`. 2000.

[100] Sun. *Sun Java Studio Creator*. Available at URL: `http://www.oracle.com/technetwork/articles/java/jscoverview-135211.html`. 2004.

[101] Gregory Tassey. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. NIST Planning Report 02-3. National Institute of Standards and Technology (NIST), 2002.

[102] Bert Thompson and Lee Naish. *A Guide to The NU-Prolog Debugging Environment*. Tech. rep. University of Melbourne, 1997.

[103] 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.

[104] Qing Yi, Vikram Adve, and Ken Kennedy. “Transforming Loops to Recursion for Multi-Level Memory Hierarchies”. In: *Proceedings of the 21st ACM-SIGPLAN Conference on Programming Language Design and Implementation (PLDI’00)*. 2000, pp. 169–181.

[105] Andreas Zeller. “Yesterday, My Program Worked. Today, It Does Not. Why?” In: *SIGSOFT Softw. Eng. Notes* 24.6 (1999), pp. 253–267. ISSN: 0163-5948. DOI: `10.1145/318774.318946`. URL: `http://doi.acm.org/10.1145/318774.318946`.

Appendices

Appendix A

Glossary of Acronyms

AD (Algorithmic Debugging): A technique to debug programs. It focuses on what is happening instead of how it is happening. To do so, AD generates an ET and asks the user questions about the information in the ET nodes. Each user's answer determines whether a node is Correct (its associated execution behaved as expected) or it is Wrong (it did not). A bug is found when a buggy node is detected (i.e., a wrong node whose children, if any, are correct).

AD search strategy: An algorithm whose input is an ET and whose output is one node of the input ET. The returned node can only be either a node whose question has not been answered yet or a node detected as buggy (i.e., the related execution produces an error).

D&Q (Divide & Query): An AD search strategy. It selects the node that better divides the tree in two parts with the same weight.

DbQ (Divide by Queries): An AD search strategy. It selects the node that better divides the tree in two parts with the same amount of remaining questions.

DDJ (Declarative Debugger for Java) An algorithmic debugger that is written in and debugs Java code.

EF (Execution Forest): A tree whose non-leaf nodes are ET nodes, and leaf nodes can be either ET nodes or EFs.

ET (Execution Tree): A tree that represents the execution of a program with a given input. Each node of the tree is associated with a concrete execution of a method of the whole execution. The root node is associated with the main procedure of the program and it is normally marked as Wrong by default.

GUI (Graphical User Interface): A type of user interface that allows users to interact with the program through graphical icons and visual indicators (definition extracted from Wikipedia).

HD (Hat-Delta) An AD search strategy. It is a TD-based search strategy that takes into account the oracle's previous answers.

HDJ (Hybrid Debugger for Java) A debugger that combines three debugging techniques, namely: Trace Debugging (TrD), Algorithmic Debugging (AD) and Omniscient Debugging (OD).

HF (Heaviest First): An AD search strategy. It is a TD-based search strategy that selects the heaviest node.

JDBC (Java Database Connectivity): This API is the industry standard for database-independent connectivity between the Java programming language and a wide range of databases, SQL databases and other tabular data sources, such as spreadsheets or flat files. The JDBC API provides a call-level API for SQL-based database access (definition extracted from the official website).

JDD (Java Declarative Debugger): An algorithmic debugger that is written in and debugs Java code. It is the predecessor of DDJ.

JPDA (Java Platform Debugger Architecture): It consists of three interfaces designed for use by debuggers in development environments for desktop systems. The Java Virtual Machine Tools Interface (JVM TI) defines the services a VM must provide for debugging. The Java Debug Wire Protocol (JDWP) defines the format of information and requests transferred between the process being debugged and the debugger front end, which implements the Java Debug Interface (JDI). The Java Debug Interface defines information and requests at the user code level (definition extracted from the official website).

JVM (Java Virtual Machine): It is the cornerstone of the Java platform. It is the component of the technology responsible for its hardware- and operating system-independence, the small size of its compiled code, and its ability to protect users from malicious programs. The JVM is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at run time (definition extracted from the official website).

LE (Loop Expansion): An AD technique that transforms the source code in order to allow AD to obtain a more debuggable ET. The technique transforms loops into their respective equivalent recursive methods.

MET (Marked Execution Tree): An ET that keeps the state of each node. The state of a node can be, among others: Undefined, Wrong, Correct, Trusted, Unknown, Inadmissible.

OD (Omniscient Debugging): A debugging technique to debug programs. It allows for inspecting the execution of the program in a forward way (as in TrD) as well as in a backward way.

OD&Q (Optimal Divide & Query): An AD search strategy. It is a D&Q-based search strategy that improves the selection of the node that better divides the tree in two parts with the same weight.

ODJ (Omniscient Debugger for Java): An omniscient debugger that is written in and debugs Java code.

SS (Single Stepping): An AD search strategy. It selects the node following a post-order traversal of the ET.

TB (Tree Balancing): An AD technique that transforms the ET in order to obtain a more debuggable ET. The technique inserts new nodes in the ET that summarizes the behaviour of a set of sibling nodes.

TC (Tree Compression): An AD technique that transforms the ET in order to obtain a more debuggable ET. The technique removes nodes related to recursive calls to avoid repeating similar questions.

TD (Top-Down): An AD search strategy. It selects the leftmost child node of the last node whose associated answer was Wrong.

TrD (Trace Debugging): A debugging technique to debug programs. It allows for placing break-points in the code, indicating the debugger must stop the execution of the program when they are reached, and inspecting the value of the variables at that point.

VET (Virtual Execution Tree): An ET where some nodes are not included because their related codes have not been executed yet.