Contracts and Slicing for Safety Reuse

Sérgio André dos Santos Areias

Supervised by:

Pedro Rangel Henriques
Daniela da Cruz

Braga, October 28, 2010

Resumo

Este documento, elaborado no Departamento de Informática da Universidade do Minho, é uma dissertação no âmbito do mestrado em engenharia informática nas áreas de reutilização de código, análise de código e slicing.

O trabalho aqui descrito focou-se no estudo e na análise de software e dos seus componentes para melhorar as técnicas de reutilização. No desenvolvimento de software, é normalmente preferível reutilizar componentes em vez de o construir de raiz. Neste contexto, consideramos que reutilizar componentes anotados é uma forma rigorosa de assegurar a qualidade de um sistema em desenvolvimento. O trabalho realizado também se focou no estudo de técnicas para slicing a fim de garantir que a integração de um componente anotado, com um contrato, num sistema irá preservar o seu comportamento.

Todo este trabalho teve em vista provar a tese que consiste em afirmar que a técnica de Caller-based Slicing aplicada a componentes com contrato permite um desenvolvimento seguro com base na reutilização. No fim, desenvolveu-se uma ferramenta como prova de conceito para testar as nossas ideias. A ferramenta permite identificar num dado sistema os componentes anotados verificando para cada um deles se as invocações respeitam as pré-condições

Abstract

This document, written in the Informatics Department of University of Minho, is a dissertation in the context of the Masters in Informatics Engineering in the areas of code reuse, code analysis and slicing.

The work here described was focus on the study and analysis of software and its components to improve reuse techniques. In software development, it is often desirable to reuse components instead of build it from scratch. In this context, we consider that reusing annotated components it is a rigorous way of assuring the quality of a system under development. The work reported also focused on the study of slicing techniques in order to certify that the integration of an annotated component with a contract, into a system, will preserve the behavior of the former.

The objective of the research done was to prove that the **Caller-based Slicing** *applied to components with contracts allows a safe development based on reuse. At the end, we developed a tool as proof of concept to test our ideas. The tool allows to identify in a given system the annotated components, verifying for each of them if the calls respect its preconditions.*

Acknowledgements

Sometimes there is the need to have something in your to help you to identify who are the real friends you have that are always by your side. Would be impossible for me to finish this work, with these results, if I did not have special friends like I have. I am grateful for that and I want to express my deepest feeling of gratitude to my friends and family, one by one.

To Professor Pedro Rangel Henriques, I am very grateful for all the support and help given through these two years of work, and for the great contribution for my academic results. Above all, thanks for not letting me deviate from my path, and for stimulate my interest in my work. His encouragement helped me to not have me essentially sloppy or even given up the thesis. More than a teacher, I consider him a friend. Muito Obrigado por tudo professor.

I would like to thank Daniela da Cruz for her enormous help in the development of the tool. Without her support I would not be able to finish the tool in time. The more time I spend with her, the more she surprises me with her immense knowledge and reasoning ability. It is impossible for me to quantify all the help she gave me in this work. Muito Obrigada Daniela.

To Nuno Oliveira, for his presence and support in carrying out my tasks. For his advice, suggestions and concerns. He has always a kind word to say, and I will never forget the funny moments we had in that lab. Obrigado Nuno.

To Daniela Fonte, for her contribution, support, availability and help in the conferences held throughout the implementation of this project. Obrigada Daniela.

To the Taekwondo team of ABC for having always supported me. With them I have improved my maturity, strength and concentration and that was reflected in the development of my work and its concretization. A special thanks to Coach Hugo Serrão, who always gave me great support, and always motivated me to achieve my goals. Muito obrigado Hugo, Rita, Nuno, Rui e Zé.

I would also like to thank my team mates from the 2004/2005 class. For all the great experiences that we had together, and for the unforgettable moments we had during this long walk. I will never forget you guys. Obrigado pessoal.

To Bárbara Bandeira, who made everything more special. Without her I certainly would have the same project and the same obstacles, but her company and fellowship made everything easier and smoother until the final straight. I will not thank you for existing but for being by my side since the beginning. Love you so much. Muito

Obrigada Bárbara.

Last, and not least, to my parents and brothers for the unconditional support, patience and motivation that helped me dealing with the stress. They have never gave up of supporting me along these six tough years and sometimes they were even more confident than me about the academic success.

I am particularly grateful to my brother Hugo Areias that helped me a lot and showed to be always available to help me overcome any difficulty. There would be many more things that I could just say thanks, but I would be here eternally; so I just want to thank you for always being by my side. Pai, Mãe e Irmãos, Obrigado por tudo.

Contents

List of Figures

Listings

Chapter 1

Introduction

The software industry has suffered big changes in the last two decades and the number of companies working on this business domain is growing from year to year. Every year a significant number of software systems is developed or improved. The reuse of software components was found to be a way to ease the development process, and at the same time reduce the costs associated to it. It is widely believed that this technique will enable software developers to create bigger and more complex systems with guarantee on the its quality and reliability. Despite the failures to introduce reuse as a systematic process on the software development, many efforts have been made to achieve this goal.

The decision to reuse raises a spectrum of issues, from requirements negotiation to product selection and integration. In order to reuse it is important to know how to choose the component that best fits respecting system requirements. According to the literature, selection of reusable components has proven to be a difficult task [MS93]. Usually, this is due to the lack of maturity on supporting tools that should easily find a component on a repository or library [SV03].

Also, non experienced developers tend to reveal difficulties when describing the desired component in technical terms. Most of the times, this happens because they are not sure of what they want to find [SV03, SS07]. Another barrier is concerned with reasoning about component similarities in order to select the one that best fits in the problem solution; usually this is an hard mental process [MS93].

Integration of reusable components has also proven to be a difficult task, since the process of understanding and adapting components is hard, even for experienced developers [MS93].

A strong demand for formal methods that help programmers to develop correct programs has been present in software engineering for some time now. The Design by Contract (DbC) approach to software development [Mey92] facilitates modular verification and certified code reuse.

The contract for a component can be regarded as a form of enriched software documentation that fully specifies the behavior of that component. So, a well-defined annotation can give us most of the information needed to integrate a reusable com-

ponent in a new system, as it contains crucial information about some constraints safely obtaining the correct behavior from the component.

In this context, we say that the annotations can be used to verify the validity of every component invocation; in that way, we can guarantee that a correct system will still be correct after the integration of that component. This is the motivation for our research: to find a way to help on the safety reuse of annotated components.

Motivated by these promises of reuse and *design-by-contract* we propose to develop a tool that allows us to to verify whether a concrete calling context preserves the precondition of the reused component, and display, whenever possible, a diagnostic or guidelines to correct any violation found during the verification. To achieve the objectives listed above, the tool implements the **caller-based slicing** algorithm, that takes into account the calls of an annotated component to certify that it is being correctly used.

This dissertation is organized as follows: In the second chapter will be addressed software reuse: the state-of-the-art, examples of real applications, issues, practical approaches. In the third chapter will be presented the concept of *design-by-contract* and few proposals using this technique. The fourth chapter will address slicing technique and its need for this research work. The fifth chapter presents the tool proposed, the sixth presents how the tool was developed, the seventh a few case studies and the eight the tool assessment. In the ninth, and last chapter, the conclusions about the research work done are drown.

Chapter 2

Software Reuse on Software Development

In this chapter we will address software reuse, its current state, its barriers, the work that still can be done and its importance on the development of new software.

The research on the area of reuse has being done for many years and it is believed that its birthday was in 1968 in the NATO Software Engineering Conference.

Software reuse is defined has the process of software development using the knowledge or components from software previously developed. It is widely believe that this can allow considerable decreases on the development cost and also on the assembling and integration efforts [FK05, SS07].

The most common benefits in the literature are the increase of productivity, quality and reliability [FK05, RDKN03, SS07]. The importance of these benefits on the future of the software development lead reuse research to quickly spread, increasing substantially the number of contributions in this area.

Usually, organizations operate in a specific business domain. This fact, makes most of their systems to be variants from others previously developed. This leads to a growth of confidence on reuse benefits, during the development of new systems.

Even being aware of reuse advantages, it is far from its real potential. A few studies have been done in order to realize (i) the best way to apply reuse on the development of new software increasing the efficiency, and (ii) why a seemingly simple concept is so complex in practice.

2.1 State of the Reuse

Even aware that much has been done for reuse in literature, there are still areas of reuse with lack of research and in need of progress in industry. A few approaches to help in software reuse were already presented in literature being three of the most important Component-Based Software Reuse, Domain Engineering and

Architecture-Based Software Reuse [SS07].

2.1.1   Component-Based Software Reuse

It is well known that many developed systems share similar components with others. So instead of build each system from scratch, we can assemble these components making the process of build new systems faster and more economic. This is the base idea of all systems developed that follow this approach.

Usually, programs based on this approach use repositories (some use libraries) since it is believed this is the best way to store a set of components built with the intention to be reusable. Open source is the best example of it with lot of repositories spread around the network sharing reusable code and reusable knowledge. A study made by Mockus years ago shows how important is the reuse nowadays [Moc07].

The main goal of this approach is to break the barrier that exists between the components developed with the intention to be reusable and the necessary effort by the programmer to find, understand and integrate these when developing big systems.

Retrieval Tools

A few tools to address with components retrieval will be presented, two of them in more detail. The first one is CodeFinder [FHR91, Hen97]. It is a system to help overcome the problems when retrieving software components from a repository. This tool is divided into three parts which we will address in this section.

The first is a tool (PEEL) to create the initial information structure of the repository. As the first thing to do is populate the it, then PEEL extract source code definitions and translate them to CodeFinder representation. These components do not need to be necessarily designed to be reused.

The translated information from components are Kandor objects [DBSB90] that CodeFinder will use to be able to create a repository structure and to index components. The result of this process is a repository with minimal retrieval structures and populated with Emacs Lisp functions, variables and constants [Hen97].

The second part of CodeFinder is a technique to make easier the search of components from the repository making the refinement of user queries. What this search algorithm does is, instead of retrieving an exact result to a query, it retrieves components associated with the query made by the user. Also the system retrieves a list of terms used to index the component into the repository allowing users to become familiar with the terms and incrementally refine their queries as they explore the information space (Retrieval by Reformulation) [Hen97, SS07].

The third and last part is an adapting tool to refine the component repository. As the development organization and domain evolves, the repository must structure must evolve too.

What this tool tries to do is the indexing and structure improving without compromising the stored information. To achieve its goals, CodeFinder uses unlimited aliasing and adaptive indexing techniques [Hen97].

The main problem of CodeFinder is that all tools it implements are separated from the development tools passing to the developer the initiative to begin the reuse process [Ye01, SS07].

To address this issue, in 2001 Yunwen proposed CodeBroker, a tool integrated into the development process, surpassing this reuse barrier [Ye01].

Instead of the systems waits for the user to make a query, CodeBroker runs in the background of the development environment (Emacs) and warns the user every time a reuse opportunity is identified. This system allows developers to reuse without switch contexts between development environments and the repository system, presenting a solution to the "no attempt to reuse" issue [Ye01].

Besides that, the system adapt itself to the developer using the concept of profile. It can be modified explicitly by the developer or the system implicitly captures developer's preferences and knowledge level.

CodeBroker repository consists of a Java API library and JGL library and the architecture consists in three software agents: Listener, Fetcher and Presenter. Every time the developer finishes to write a doc comment or a function definition, Listener extracts the context information and creates a concept query with it. Later this concept query is passed to the Fetcher and Presenter. Fetcher is responsible for the retrieving of components from the repository. These components must show high conceptual similarity to the content of the doc comment or high constraint compatibility to the function definition. Presenter gets these retrieved components and display them to the user in the RCI-display. The user profile is taken in consideration when doing it.

2.1.2 Domain Engineering and Software Product Lines

This approach is based on the reuse of domain knowledge to develop new systems with an improve of quality, and has a key role in systematic reuse. Unlike Component-Based Software Reuse, reusable assets are not created and stored waiting for a reuse opportunities to be used, but instead they are only developed when commonalities and variabilities are identified in developed systems.

As already seen, most of the organizations operate in a specific domain and that makes most of the developed systems variants of others previously developed. Based on this fact, it is widely believed that this approach can significantly decrease the time-to-market and the costs while increases productivity. A study made by Batory et al. [BJMvH02] shows the substantial decrease on the effort needed to develop a new system.

Domain Engineering is divided in two phases, domain analysis and domain implementation. Domain analysis is the process of analyze the set of systems of a domain

to identify the commonalities and variabilities [FI94, SS07]. This information is represented under the form of a domain model. Domain implementation is the use of that knowledge to develop reusable assets in order to use these in the development of new systems.

Domain analysis depends on the domain knowledge owned by the organization and the experience of the staff to analyze this information and design reusable artifacts.

A few systems based on this approach were developed and have shown some progress on software reuse [SS07, FK05].

2.1.3 Architecture-Based Software Reuse

This approach begun to gain more supporters in the 1980's because software architecture have shown to have a strong impact on system quality attributes [FK05].

Software architecture is the combination of components, their external properties, and the way these are related to each other. This is the base to architecture-based software reuse assets [SS07].

In [FK05] Frakes and Kang have shown that a software architecture presents different levels of abstraction. Architecture styles at the lower level. The combination of these allows the creation of architecture patterns. Examples of both can be found in [Sha95] and [BMR$^+$96] respectively. As seen in other approaches, also here systems in the same domain share commonalities, in this particular case they share architectural patterns. The combination of these similar patterns allows the creation of a generic architecture, i.e., a domain architecture. Examples of domain architectures can be found in [Tra95]. Using these domain architectures, application architectures can be built. These have the higher level of abstraction.

2.1.4 Reuse in Industry

A study made showed that some groups had made some progress with some aspects of reuse, but this still not a common practice on the software development process [Gri91]. Examples of companies that are trying to achieve the benefits of systematic reuse are: IBM, Microsoft, Motorola, Siemens, Hitachi, Fujitsu, Toshiba [FI94].

Hewlett-Packard is an example of a company that adopted reuse and one of the first to do so and showing some progress [MET02].

HP showed interest in understanding and applying all the knowledge about reuse to include it as a systematic process in their production lines, and try to achieve the true potential of reuse [Gri91]. The main idea was to make a thorough investigation to later be possible to develop a system to support reuse practices on the company.

They believe this system can have benefits if based in hypertext concept. With this tools, HP experts expect to be possible for developers to search, evaluate and integrate reusable components in a quick and efficient way [CFG91].

HP developed one prototype to achieve all these ambitions [CFG91]. Kiosk is a system developed in C++ also using InterViews for the user interface [CFG91, Tsi91, Gri91].

Kiosk

Kiosk is a system that stores libraries of reusable software components and also allows the management of those.

To browse libraries contents, Kiosk provides a browser so developers can navigate and interact with the components. Every component is represented as a node and every relation between them is represented as links.

To import reusable components to Kiosk libraries, it uses a tool called Cost++ that uses the specifications included in the properly input. The input data specifies how the library in question will be structured. The Cost++ result consists in a classification network formed by the following three type of nodes: classification nodes, spoke node, hub nodes [Tsi91]. This makes possible, to the user, to view the content of a library from different views (the system provides four different type of views). With a variety of views, users can choose the one that suits better for a particular task.

Another feature of Kiosk is the possibility for users to create their own annotations. This annotations can be seen when browsing components of the library helping users to understand the component whithout having to study the documentation or even the interface description.

This system uses the Full-Text technique to retrieve components from the library. As Kiosk, like we saw before, stores more than one library, the search can be tedious. To avoid this problem the system allows the use of regular expression making possible to the search algorithm to use, per example, links properties in the match. The results from the search are shown as an hypertext structure allowing the user to interact directly with the work product or with the components or classifications where that work product belongs [Tsi91].

After the first release of Kiosk a questionnaire was promoted by HP to receive a feedback from users. The installation of the system proved to be simple but most of the users complained about the lack of documentation and support. The study proved that Kiosk can be useful but not mature enough to be used by novice users [Tsi91].

2.2 Reuse Barriers

During the history of reuse were pointed several obstacles to apply it. A few critical factors to achieve a reuse with success were being discussed in literature but while there is agreement in some factors there are some that still need research and a few more discussion. Based on past experiences of reuse, it is believed that top man-

agement and software developers are the first barrier to reuse adoption on industry [SV03]. Reuse barriers were divided in four main groups as is believed that these will include the more critical barriers to reuse adoption.

2.2.1   Top Management

An important factor is that nontechnical issues of reuse were mainly introduced to discussion in the last decades of research since previously only technical issues were addressed [MET02, MS03]. Some research about this is still needed and the need of approaches to support this research is big [Dav93].

Based on past experience, one of the main difficulties is getting companies to use the reuse as a systematic process in their production lines [FK05]. Sherif and Vinze study clearly shows that management unwillingness to invest on reuse is one of the primary barriers to its success.

From a company point of view, reuse does not cease to be an investment, and as such can not be seen as an academic research. To be susceptible of an investment it is necessary that there are guarantees about reuse efficiency and reliability to present financial benefits on software development. Moreover reusable components, due to the bigger level of abstraction they show, require more resources during the development [Kru92].

The way these investments are made varies from company to company. Companies with high levels of organization and with a total control of future requirements usually would prefer an initial investment while other companies only do it when appears an opportunity to reuse during the development. These decisions are important but, regardless the one chosen, it is necessary to make an investment since without it will be difficult to reach the so expected systematic reuse.

With the problem of having to develop fast and with low costs, usually makes management to not show much willingness to invest substantial amounts in long-term benefits with almost no short-term returns [SV03, MS93, Fav91, FI94]. The concern to develop fast to deliver fast, increases management unwillingness to allocate resources to systematic reuse and diverts the attention away from the possible benefits and additionally, clients show interest in the final product and if that will be delivered on time and not really on reuse concept [SV03].

Also a way to efficiently measure the benefits of reuse on development is needed as the lack of it can be an aversion for management adoption of reuse [SV03, FK05].

Rothenberg et al. [RDKN03] made a study that shows that organizations that support reuse and use it as an integrated part of the development process, can have the full benefit of it. While most of the problems and risks associated to reuse persist then, more companies will still avoid to use reuse as a repeatable process. In history some companies made great efforts to reuse and received disappointing results [Dav93].

There is no recipe to implement the reuse process on a company. Management must

analyze organization's needs and adapt reuse taking the results in consideration. Companies might benefit with the reuse experience of other companies with higher levels of reuse [FI94, FF95].

2.2.2   Technical and Organizational Barriers

To achieve a systematic reuse is necessary to have software reuse as a systematic process in the development methodology. Most of the problems arise when trying to integrate reuse in the traditional development methodology of an organization. A study made by William Frakes and Christopher Fox [FF95] shows that if the reuse is treated as an integrated process in the development of new systems the reuse levels are higher. Along time, some proposals were made in literature to fulfill this necessity [Sim88, Kan88].

Some experts points that object-oriented (OO) technology is essential to reuse and itself can assure the creation of reusable assets. It is clear that OO approach has some important properties for reuse practice. For example, encapsulation and the fact that some techniques are provided to allow the use of artifacts in several projects, forces the developer to take into account OO analysis and design. This will give, to the developer, a view of the problem in terms of the businesses process [RDKN03].

In 1995, William Frakes and Christopher Fox realized that programming languages do not affect reuse levels [FF95]. Another study made by Karma Sherif and Ajay Vinze contradicts some experts presenting that OO technology itself is not enough to guarantee reuse and sometimes can be even a barrier to it [SV03].

Also is necessary the existence of a reuse group with autonomy to improve reusable components, reuse process and work on supporting tools. HP is an example of a company with positive results with an autonomous reuse group [Gri91].

During the history of reuse also some technical barriers were gaining importance. The process to build reusable assets and the support to store, use and maintain the assets as shown to be a hard task to put in practice. The design of a component can be a very complex task. Reusable components can not be simply developed with the intention of being used only in a particular case. These must present a high level of abstraction in order to be flexible to future needs [SV03, Kru92]. Also, the creation of documentation for reusable components can be a time consuming process and sometimes avoided. WIth lack of detailed components' documentation, will increase the difficulties in components' understanding and this can be a limitation to the use [SV03].

A well-defined reuse policies to cover all this issues could be a solution to increase reuse levels on an organization and break these barriers.

The instability of the technology is believed to be one of the reasons to companies drop the support on systematic reuse. Describe, understand, select and adapt assets are complex tasks and the need of tools to help is imperative [MS93]. Even so the same study made by William Frakes and Christopher Fox shows that retrieval tools

do not have a significant impact on reuse levels.

Retrieval Tools

It is known that to support developers with a good reuse, organizations need to provide repositories containing enough components. This makes more difficult the process to search and select the right component, also bringing scalability problems. Is then necessary to have a minimally structured repository and indexed because without this will be very difficult to achieve a good retrieve [Hen97, SS07]. Due to this need, some techniques were proposed to assist in retrieval algorithms [FG90]

The difficulty in ensuring a good scalability and the constant evolution on the business domain of a company makes difficult to obtain a dynamic structure on a repository. With this, any change in the context of the repository will be a quite complex effort [Hen97].

To a developer be able to find, understand and integrate a component in the new system it is imperative that all components, on the repository, have an associated abstract specification [Kru92]. This will make possible, to the developer, to reuse without having to study the component source code. Also the time to reuse will not exceed the time to build the component from scratch.

One of the biggest problems in retrieval tools is that, usually repositories indexation are made by expert using terms in the classification of components that are not common to non-expert users. This shows to be an issue because usually most of the users only have vague idea of what they want to find making more difficult to express in a well-defined query [Hen97, SS07]. These differences in the terms, used to classify components, are common when the indexation is manually and some studies prove that [Hen97].

A study made by Karma Sherif and Ajay Vinze [SV03] shows a few problems that developers face when trying to use components from a repository, specially when trying to find or when trying to integrate a component.

2.2.3   Measurement of Reuse

Being software reuse an area to be based on science and engineering then it is necessary to know how to measure concepts like reusability [FK05].

During the reuse evolution, metrics and models have been defined for many areas of reuse [FT96, SS07]. Even so there is still a lack of research in this area of reuse.

A study made by Frakes and Fox [FF95] and other made by Sherif and Vinze [SV03] show consistent findings; reuse measurement does not affect reuse levels. The true is that most of organizations do not make any attempt to measure reuse benefits (improvements on productivity and quality) and cost.

Even if not affecting reuse levels, reuse measurement is imperative to systematic

reuse [FI94, FF95]. This can be used to convince management of the possible benefits of the software reuse [MMM95, SS07]. Reuse measurement will also make possible to compare and choose the best reuse strategy. Mili et. al [MMM95] reflects that without well-defined metrics can be no objective basis for comparing different reuse approaches.

During the development process many decisions must be taken. Mili et. al [MMM95] reported three types of decisions. The decision an organization must take to launch a software reuse program, the decision to develop reusable assets and the decision to integrate a reusable asset in a new system. All these decisions will have implications and therefore these must be measured [Car89, MMM95].

Reuse measurement is a practice that has to be taken seriously by organizations. Traditional software metrics must be amended to include software reuse variables. Only that way will be possible to measure reuse contribution to the productivity, to the development process, to the developers effort, to the quality, etc. [MMM95].

2.2.4   Human Issues

Human involvement was ignored as a barrier to the adoption of software reuse for many years. In the later years, a few studies have shown that a few problems may show up when ignoring the role of software developers in the reuse process [MS93, SV03].

For many years experts regarded as one of the biggest problems on software developers the "Not Invented Here" syndrome. However later studies concluded that most of the times the opposite happens, i.e., software developers are willing to reuse instead of develop a system from scratch [FF95, Fav91].

For us it is clear that one of the biggest problems is the lack of reuse education shown by software developers. To this issue be overcome it is necessary to have some points in consideration. Component comprehension is a consuming task even for experts and it is not easy to introduce this type of education in an environment ruled by tight deadlines [MS93].

Experts show better mental representations what make them to better identify opportunities to reuse and identify system's commonalities in a domain than less experienced developers [SV03, MS93]. These experts can have influence in organization's reuse levels [SV03] and therefore is important to give proper training to the staff. A study made years ago [FF95] have shown that educated developers can present significantly higher median levels of reuse. Even aware of that, this type of education continues to be not so popular within academic and industry.

2.3   Summary

In this chapter we have presented the concept of software reuse. A simple idea but at the same time so hard to put in practice. We have also described the three major approaches to apply reuse: Component-Based Software Reuse, Domain Engineering and Software Product Lines and Architecture-Based Software Reuse. A few tools that support these approaches were also presented.

In this chapter we have also shown and discussed the possible benefits of reuse and the barriers for its integration on the software development process to achieve systematic reuse.

Chapter 3

Design by Contract

The reuse is a very important feature to the Object-Oriented technology. As seen in the previous chapter, it is important to guarantee the reliability of a system with reused components. They can be integrated in thousands of different systems increasing the risk of possible failures [Mey92]. There is a strong belief that Design by Contract(DbC) can increase the reliability and bring several advantages to the process of software development [Fel03, Mey92, LC03, TBmJ06, JM97].

In this chapter we will enumerate this advantages and present some successful cases of the DbC application. We will also present this important concept that is related with our work.

DbC is a designing approach for the construction of software introduced by Meyer in 1986 [MNM87]. The main idea behind this technique is the development of components together with their specifications.

The contract works in a similar way as a human contract between a client and the supplier. It includes the benefits and obligations for both sides in order to guarantee a correct call (code context). Usually an obligation for one party implies a benefit for the other [Mey92].

These contracts are, usually, formed by assertions (preconditions, postconditions or invariants) that describe the requirements and the return conditions of a software component [Mey92, LC03, dCPH09].

If this assertions are not respected, it means that one of the sides have not fulfilled its contract obligations. We can identify the violator using the type of the violated assertion. If it is a precondition violation then it was the client not respecting the supplier requirements. If it is a postcondition violation then it was the supplier not delivering the agreed return.

The contract theory works in the opposite way of the idea of defensive programming. While the second encourages the development to protect all software modules checking blindly as many times as possible, adding redundancy and increasing software complexity, the first declines a contract that assigns the responsibility for every consistency condition both parties (client and supplier) [TBmJ06, Mey92].

The use of assertions would make unnecessary the use of the redundant checks included in the software packages developed using defensive programming. This is just one of the advantages. We will see more in the next section.

In 1986, Bertrand Meyer designed a language with native support for DbC, Eiffel [MNM87]. This was the first practical application of it.

3.1   Contract Advantages

Along the years many reasons were presented to justify the use of DbC. In this section we will present a few reasons to use it when developing software.

There is a strong belief that the use of contracts can help during the phases of testing, debugging, maintenance and a few others [NE02, Mey92].

As the contracts are defined using the programming language itself, they can be translated by the compiler to executable code. This allow us to do the contract verification in runtime making possible to find faults while the program is running. If a fault is found, an exception can be thrown or we can even run a recovery code to put the program in a safe state again [TBmJ06, LC03]. The diagnosis scope of these faults is significantly smaller due to the assertions on the contract, i.e. when an assertion is violated we know exactly the location of the fault reducing the time to identify it [TBmJ06, Mey92, LC03].

The fact that the contracts are executable also can give a big contribute to the test phase. Leaving results verification to the contracts, the tests suffer a reduction of code needed. Also, if a change is made to the code there is no need to change the tests, but only the contract, what is less consuming [Fel03].

Contracts also improve the understanding of a program, because we can read how the program will behave and what it requires to produce the correct result [LC03].

Yves Le Traon et al. [TBmJ06] made a study that shows the impact of DbC in the vigilance and diagnosability of a system. They have shown that the use of contracts can help to increase the software robustness, and allows an early detection of the faults and their location (Figure 3.1). Also, they have shown that the diagnosability and quality of a system also increases. This increase has no relation with the number of contracts, but instead with their efficiency.

In 1997 Meyer et al. [JM97] used as case study the accident of the maiden flight of the European Ariane 5 launcher to alert to the importance of DbC in the software development process. Under the various analysis that they do, they state that probably the accident would not happen if the use of the contract was present. The check of the obligations satisfaction by each client (call) would have caught the error and launch an exception probably avoiding a 500 million accident.
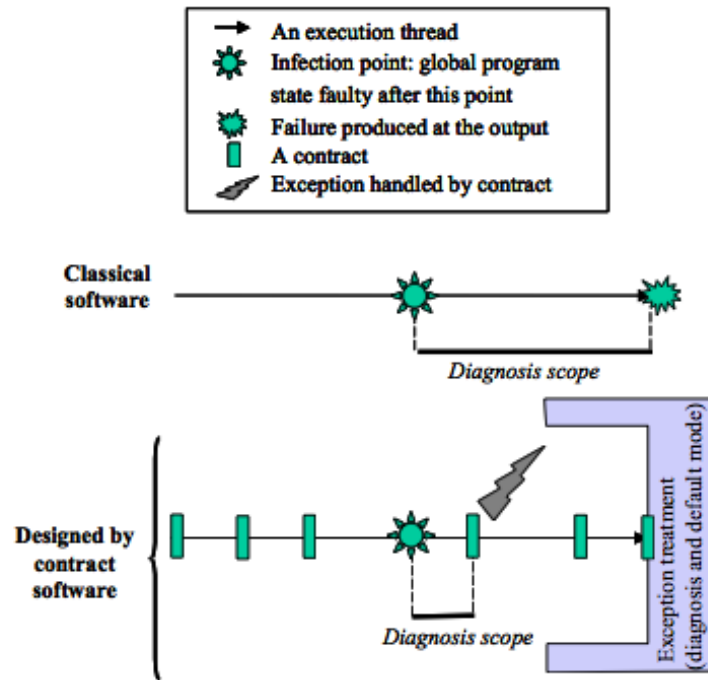
Figure 3.1: Contracts for early detection of a fault (taken from [TBmJ06])

3.2 Applying Design by Contract

For many years techniques to support DbC were developed [WPF$^+$10, FLRL00, NE02, DF06]. One of the most important type of tools developed are those that do inference of specification from the source code. A well-known tool for dynamic inference of invariants is the Daikon [Ern00, ECGN99].

As any other dynamic approach, Daikon relies on the quality of the tests to generate the candidates, what makes it not totally reliable [DF06]. Even so it presents positive results as shown by Nimmer and Ernst in [NE02]. They have shown that even with a few limitations, the specifications dynamically inferred by Daikon are reliable enough to be machine verifiable.

Another important tool is the ESC/Java [DRL$^+$98], but to make static verification of annotations. This is one of the most capable tools to make modular checking of executable code.

ESC/Java presents many limitations and only detects a few types of errors [NE02]. One of the reasons that leads to avoid use of ESC/Java by the developers, is the fact that leaves to them the responsibility to write the annotations. This was one of the reasons that lead to the development of Houdini [FLRL00].

Houdini is an annotated assistant for ESC/Java. Analyzing the executable code, it generates a set of candidate assertions referred from the unannotated program, and uses ESC/Java to check them. This is done as many times needed to reach a fixpoint. Each cycle means that ESC/Java found invalid annotations that are

immediately removed by Houdini from the program. This result is then given to ESC/Java and a new cycle begins.

The use of Houdini decreases significantly the time needed by ESC/Java to check an unannotated program, and also makes the result more reliable as reduce the number of false alarms by ESC/Java. All this as a pay off. In exchange, Houdini consumes lot of computing effort, and that is why it is used mainly for debugging instead of being used for code generation with safety certification.

In [DF06], Denney and Fischer concluded that with a new approach they can improve the results of this kind of tools. Their approach presented better results in respect to flexibility, extensibility and necessary effort. This even with a few deficiencies in their system.

In the last years, has been a considerable progress in this area. An example of it is the AutoFix project, particularly one of its components, the tool AutoFix-E [WPF$^+$10] that is somehow related to our work.

Basically, this tool proposes solutions to a software fault and validates them relying on the contracts present on the software.

Its results have shown to be very promising. 10 data structures classes were used to the experiments. 42 faults were detected and from those 16 were fixed automatically Several of those fixes were identical to fixes proposed by experts.

3.3   Reuse by Contract

This concept appears in sequence to Meyers definition of DbC. It is defined as the application of formal methods to software reuse [FS97].

This idea aiming at making reuse negotiations between the client and the supplier based on the use of contracts.

Reuse by Contract suffers practically from the same issues as the reuse and the DbC. Usually, it can be seen as the same concept as DbC, but that is a mistake taking into consideration the role that the contract plays in each technique [FS97]. Fischer and Snelting [FS97] strongly believe that only integrating Reuse by Contract in the systematic process of software development will justify its name. To quote them:

> In the design approach, contracts are passive and confined to the library. They do not only describe the components properties but are also their possessions. The reuse approach removes this asymmetry and "activates" the contracts for prospective clients. It is a way to exploit the full power of contracts.

Most of its benefits were already seen in the software reuse chapter (Chapter 2). Some examples are: better matches when retrieving components, and integration of components without compromising the correctness of the system.

Along time a few tools and approaches to support this idea emerged. Some to address the issues of structuring a library and retrieve components from it [RW91, SF97, SW94, MMM97] like the one proposed by Jeng and Cheng [JC93]. They have proposed a classification scheme of software components to allow the reuse of software components based on formal specifications. They also show some algorithms to construct a two-tiered hierarchical library from the formal specifications in order to ease users with the selection and retrieving of reusable components. Also approaches to support the specification and verification of programs were proposed [Lea91]. Zaremski and Wing [ZW96] proposed an approach based on specification matching in order to compare software components. With this they were able to verify if a component can be replaced by another or even how it can be modified to fulfill another component requirements. To support this idea they have also developed a tool.

3.4   Summary

In this chapter we have shown in what consists the Design by Contract theory, and in what it can be very helpful. We also presented several of its advantages described in literature. A few tools that support this technique were presented and characterized.

Besides this technique, we have also discussed the Reuse by Contract concept. We presented some of the differences to DbC, and discussed a few of its advantages. An the end we have described some tools based on this philosophy of software development.

We will end this chapter with a statement that defines our opinion and belief about DbC. In [JM97] Jézéquel and Meyer conclude

> "Effective reuse requires Design by Contract. Without a precise specification attached to each reusable component - precondition, postcondition, invariant - no one can trust a supposedly reusable component. Without a specification, it is probably safer to redo than to reuse."

Chapter 4

Slicing

Program slicing is a method of program decomposition proposed by Weiser in 1979 in his PhD thesis [Wei79]. Along time, program slicing has proved to be helpful in many aspects of the software life cycle, including software testing [Bin99, HD94], software metrics [OT93], software maintenance [GL91], program comprehension [dLFM96, HHP+01], reuse of software components [BE93, CDLM95], program integration [BHR95, HPR89] and so on. We will present some of these approaches later in this chapter. It is defined as the decomposition of a program into a smaller that contains all statements relevant to a particular computation [Wei81, AdCP10]. A program slice consists of the parts of the program that potentially affect the values computed at some point of interest referred to as a slicing criterion (see Definition 1).

A slice criterion can lead to one or more different slices from a program (the program itself).

Definition 1 (Slicing Criterion)**.** *A static slicing criterion of a program P consists of a pair $C = (p, V_s)$, where p is a statement in P and V_s is a subset of the variables in P.*

A slicing criterion $C = (p, V_s)$ determines a projection function which selects from any state trajectory only the ordered pairs starting with p and restricts the variable-to-value mapping function σ to only the variables in V_s.

Definition 2 (State Trajectory)**.** *Let $C = (p, V_s)$ be a static slicing criterion of a program P and $T = < (p_1, \sigma_1), (p_2, \sigma_2), ..., (p_k, \sigma_k) >$ a state trajectory of P on input I. $\forall i, 1 \leq i \leq k$:*

$$Proj'_C(p_i, \sigma_i) = \begin{cases} \lambda & if\, p_i \neq p \\ < (p_i, \sigma_i | V_s) > & if\, p_i = p \end{cases}$$

where $\sigma_i | V_s$ is σ_i restricted to the domain V_s, and λ is the empty string.

The extension of $Proj'$ to the entire trajectory is defined as the concatenation of the result of the application of the function to the single pairs of the trajectory:

$$Proj_C(T) = Proj_C^{'}(p_1, \sigma_1)...Proj_C^{'}(p_k, \sigma_k)$$

A program slice is therefore defined behaviorally as any subset of a program which preserves a specified projections in its behavior.

Definition 3 (Static Slicing). *A static slice of a program P on a static slicing criterion $C = (p, V_s)$ is any syntactically correct and executable program P' that is obtained from P by deleting zero or more statements, and whenever P halts, on input I, with state trajectory T, then P' also halts, with the same input I, with the trajectory T', and $Proj_C(T) = Proj_C(T')$.*

4.1 Static Slicing

Static Slicing was firstly introduced by Weiser [Wei81]. This type of slicing compute slices only using information statically available. Its definition can be seen in Definition 3.

Three major program slicing approaches emerged overtime. The first was introduced by Weiser in his original slicing approach. It is based on iterative solution of dataflow equations [Tip94].

In 1985, Bergeretti and Carré [BC85] introduced information-flow relations, another approach that can be used to compute slices. Another alternative to these two approaches is the one suggested by Ottenstein and Ottenstein in [OO84]. This defines slicing as a reachability problem in a dependence graph representation of a program. In this approach the Program Dependence Graph, also called as Procedure Dependence Graph (PDG) as it will be called from now on, is used for static slicing of single-procedure programs. Later, Horwitz et al. [HRB90] presented an extension of PDG, the System Dependence Graph (SDG), to find interprocedural program slices.

In [Tip94, XQZ⁺05] are described a list of proposed algorithms based on these three approaches. We will present a few examples of the static slicing algorithms and present the definition of PDG and SDG because will be useful to understand our proposal later on.

4.1.1 System Dependence Graph

Many of the approaches that will be presented here are dependence graph based, including the one we use (see Section). Considering this, it is important to understand what is a PDG and a SDG. In this section we will present their formal definitions.

Definition 4 (Procedure Dependence Graph). *Given a procedure $\mathcal{P}$, a Procedure Dependence Graph, PDG, is a graph whose vertices are the individual statements and predicates (used in the control statements) that constitute the body of $\mathcal{P}$, and the edges represent control and data dependencies among the vertices.*

In the construction of the PDG, a special node, considered as a predicate, is added to the vertex set: it is called the *entry* node and is decorated with the procedure name.

A control dependence edge goes from a predicate node to a statement node if that predicate condition the execution of the statement. A data dependence edge goes from an assignment statement node to another node if the variable assigned at the source node is used (is referred to) in the target node.

Additionally to the natural vertices defined above, some extra assignment nodes are included in the PDG linked by control edges to the entry node: we include an assignment node for each formal input parameter, another one for each formal output parameter, and another one for each returned value — these nodes are connect to all the other by data edges as stated above. Moreover, we proceed in a similar way for each call node; in that case we add assignment nodes, linked by control edges to the call node, for each actual input/output parameter (representing the value passing process associated with a procedure call) and also a node to receiving the returned values.

Definition 5 (System Dependence Graph). *A System Dependence Graph, SDG, is a collection of Procedure Dependence Graphs, PDGs, (one for the main program, and one for each component procedure) connected together by two kind of edges: control-flow edges that represent the dependence between the caller and the callee (an edge goes from the call statement into the entry node of the called procedure); and data-flow edges that represent parameter passing and return values, connecting actual$_{in,out}$ parameter assignment nodes with formal$_{in,out}$ parameter assignment nodes.*

4.1.2   Static Slicing Algorithms

In the last three decades, thousands of new approaches have been proposed to improve slicing methods. We will present some of those we find most interesting.

In 1990 Horwitz et al. [HRB90] proposes an algorithm for computing precise interprocedural static slices that operates on a program representation (SDG). This can be divided in two phases. The first is the construction of an attribute grammar, that models the calling relationships between the procedures, in order to complete the SDG with summary edges. The second consists in two phase traversal of the SDG to compute the slices. This algorithm does not guarantee that the computed slices are executable programs. To address this limitation, Binkley [Bin93] presented an extension of the Horwitz et al. algorithm. Over the years, several other extensions of this algorithm have been proposed [Lak92, CFR+99, OSH01].

In [Lyl84], Lyle concluded that Weiser's algorithm for static slicing could yield incorrect slices in the presence of unstructured flow. In his PhD thesis he presented a solution for this unwanted results. A conservative algorithm that includes in the produced slices any `goto` that has non-empty set of relevant variables associated with it, thus dealing with the `goto` statements.

Gallagher [Gal89], Jiang [JZR91], and Gallagher and Lyle [GL91] also proposed vari-

ants of Weiser's algorithm with unstructured control flow. Even so, these algorithms may produce incorrect slices, as concluded by Agrawal [Agr94].

Ball and Horwitz [BH93], and Choi and Ferrante [CF94] concluded that PDG-based slicing algorithms are fallacious in the presence of unstructured control flow. These algorithms fail to determine correctly when unconditional jumps are required in a slice, and that is the problem on them.

Many other static slicing algorithms were proposed. A comparison between them can be found in [Tip94].

4.2 Dynamic Slicing

The notion of Dynamic Program Slicing was introduced by Korel and Laski [KL88]. While the static program slicing produces slices that contain all statements that are relevant to a particular computation for any input, the dynamic slicing does the same but for a specific program input.

As the role of slicing in program debugging is to reduce its effort, a result of computed large slices when debugging is undesirable. This was the motivation for this kind of slicing.

Dynamic slicing has great advantages for program debugging but, in against, it needs a significant compute effort [AH90, ZGZ03]. Due to this, imprecise dynamic slicing algorithms have been proposed along time. These algorithms simply need less computing effort at the cost of some dynamic slicing precision.

The three major approaches presented for static slicing, also apply for dynamic slicing. In the next section, we will present a few algorithms based on these approaches.

4.2.1 Dynamic Slicing Algorithms

After Korel and Laski [KL88, KL90], many proposals to improve dynamic program slicing emerged. Gopal [Gop91] introduces a dynamic version of Bergeretti and Carré [BC85]. This approach computes dynamic slices basing on dynamic dependence relations. The advantage of these relations is that may compute smaller slices compared with the original approach.

In 1988, Miller and Choi [MC88] introduced, for the first time, the concept of Dynamic Dependence Graph (DDG). This graph is a dynamic variation of the conventional PDG. In their work, they used this graph to perform flowback analysis [Bal69] on parallel programs.

Agrawal and Horgan [AH90] also based their approach to compute non-executable dynamic slices on dependence graphs. They presented three different approaches. The first two are simple and efficient and use PDG to compute the slices. The drawback of this is that makes the computed slices inaccurate. The third approach

proposed uses DDG to compute accurate dynamic slices. The problem is that, in a DDG, the number of nodes is equal to the number of executed statements. Due to the size that a DDG could have, Agrawal and Horgan proposed the Reduced Dynamic Dependence Graph to reduce it without losing computed slices precision. This solution still might have unbounded size has pointed by Tip [Tip94].

Recently Zhang et al. [ZGZ03] proposed a precise dynamic slicing approach. They have concluded that a well designed precise dynamic slicing can be a better choice than an imprecise one, pointing that it can be faster and practical. In their study, they also show that their precise dynamic slicing is faster than an imprecise dynamic slicing algorithm proposed by Agrawal and Horgan [AH90]. We already discussed those algorithms here.

In the last decade, there have been many contributions in this area. Xu et al. [XQZ$^+$05] presented a great survey where a description of these contributions can be found.

Zhang and Gupta [ZG04] proposed a cost effective dynamic program slicing to compute slices efficiently. This approach is based upon a DDG representation that is highly compact and rapidly traversable. They compare it with three precise dynamic slicing algorithms [ZGZ03](that they earlier proposed), and they conclude that their approach provides faster times when computing dynamic slices, and that the DDG is significantly more compact.

4.3 Application of Program Slicing

The original idea of slicing was to aid in the debugging process. The produced slices would contain the fault ignoring all statements that do not satisfy the slicing criterion. However, it was only a matter of time before someone realizes that slicing could be an important help in many aspects of the software life cycle.

In this section, we will show some slicing approaches proposed to help on other areas of research. A detailed information about most of these approaches can be found in [Tip94, XQZ$^+$05].

4.3.1 Debugging

Dynamic slicing is one of the most important proposals that address the debugging problem. The reduced size of slices, compared with the static slices, and the fact that the computed slices only reflect the actual dependences of a specific input (the one that produces incorrect values), makes them very useful.

Most of the approaches presented already in this chapter can be used for debugging purposes.

Another related work motivated by debugging was the one introduced by Lyle and Weiser [LW87], the program dicing. However, this approach can produce incorrect results in the presence of multiple bugs.

Comuzzi et al. [CH96] was the first to introduce program slicing based on specifications with the notion of predicate slice (p-slice), also known as postcondition-based slice. This removes all the statements that are not required for the validity of a specific postcondition upon termination.

Canfora et al. [CCL98] introduced conditioned slicing. This computes a forwarding slicing removing all the unreachable code using a set of states previously defined in the precondition.

Chung et al. [CLYK01] proposed a combination of both, preconditions and postconditions to compute slices.

Barros et al. [BdCHP10] proposes the Assertion-based slicing that combines the forward propagation of preconditions, and the backward propagation of postconditions, to improve computed slices precision in comparison with the existing specification-based slicing approaches.

Zhang [ZXG05] proposes modular monadic slicing, a formal method for program slicing. This approach compute slices without resorting to intermediate structures (ex. dependence graphs), or to record an execution history in dynamic slicing algorithms. It shows great flexibility and reusability properties when compared with any other program slicing algorithms.

4.3.2 Software Maintenance and Reuse

Slicing can also be used to address some problems on software maintenance. For example, it can help determining which parts of a program will be affected by a change.

Gallagher and Lyle [GL91] proposed an approach based on the decomposition of a program into a set of slices, where each captures all the computation for a specific variable.

Horwitz et al. [HPR89] proposed an approach for program integration. This compares slices in order to detect equivalent behaviors making easier the study of program behavior. Later, a few variants of this work were presented [RY89, Rep91].

Zhao [Zha98] introduced the concept of slicing applied to an architecture. It works in a similar way as a program slicing. The computed slice contains the components, correctors and configuration of the architecture concerning a slicing criterion. Later, Kim et al. [KtSCH99] proposed a dynamic version of this approach.

Many approaches were proposed to help with reuse [CDLM95, CDLM96, LV97]. Canfora et al. [CLLF94] presented an approach to help identifying reusable functions in a program. This approach used slicing to decompose code functions into more elementary components.

4.3.3 Program Comprehension

Usually program comprehension is applied in the early phases of software maintenance. To do the maintenance of some programs it is necessary to understand it first. Slicing can also help addressing this issue.

Some of the approaches already presented in this chapter, found to be also useful to program comprehension.

Korel and Rilling [KR98] presented several concepts for static and dynamic program slicing. Also, they combined them with several methods of visualization in order to help the developer with program comprehension. They concluded that the features proposed could be a great help for developers during the process of program understanding.

Ball et al. [BE94] presented a tool that allows interactive browsing of the computed slices.

Gallagher et al. [GO97] proposed an approach based on decomposition slices reducing that way the visualization complexity.

Deng et al. [DKN01] presented a tool called Program Slice Browser. This tool main goal is to extract useful information from a complex program slice in an interactive way.

Krinke [Kri04] presents a declarative approach to layout PDGthat generates comprehensible graphs of small to medium size procedures. The authors discussed how a layout for PDG can be generated to enable an appealing presentation. The PDG and the computed slices are shown in a graphical way. This graphical representation is combined with the textual form, as the authors argue that is much more effective than the graphical one. The authors also solved the problem of loss of locality in a slice, using a distance-limited approach; they try to answer research questions such as: 1) why a statement is included in the slice?, and 2) how strong is the influence of the statement on the criterion?

Balmas [Bal04] presents an approach to decompose SDGin order to have graphs of manageable size: groups of nodes are collapsed into one node. The system implemented provides three possible decompositions to be browsed and analyzed through a graphical interface: nodes belonging to the same procedure; nodes belonging to the same loop; nodes belonging to the two previous ones.

4.4 Caller-based slicing

In this section, we introduce another slicing algorithm. We start by extending the notion of static slicing and slicing criterion to cope with the contract of a program.

Definition 6 (Annotated Slicing Criterion)**.** *An* annotated slicing criterion *of a program* $\mathcal{P}$ *consists of a triple* $C_a = (a, S_i, V_s)$, *where* a *is an annotation of* $\mathcal{P}_a$ *(the annotated callee),* S_i *correspond to the statement of* $\mathcal{P}$ *calling* $\mathcal{P}_a$ *and* V_s *is a subset*

of the variables in $\mathcal{P}$ (the caller), that are the actual parameters used in the call and constrained by α or δ.

Definition 7 (Caller-based slicing). *A caller-based slice of a program $\mathcal{P}$ on an annotated slicing criterion $C_a = (\alpha, call_f, V_s)$ is any subprogram $\mathcal{P}'$ that is obtained from $\mathcal{P}$ by deleting zero or more statements in a two-pass algorithm:*

1. *a first step to execute a backward slicing with the traditional slicing criterion $C = (call_f, V_s)$ retrieved from C_a — $call_f$ corresponds to the call statement under consideration, and V_s corresponds to the set of variables present in the invocation $call_f$ and intervening in the precondition formula (α) of f*

2. *a second step to check if the statements preceding the $call_f$ statement will lead to the satisfaction of the callee precondition.*

For the second step in the two-pass algorithm, in order to check which statements are respecting or violating the precondition we are using abstract interpretation, in particular symbolic execution.

According to the original idea of *James King* in [Kin76], symbolic execution can be described as "instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values. The execution proceeds as in a normal execution except that values may he symbolic formulas over the input symbols."

Using symbolic execution we will be able to propagate the precondition of the function being called through the statements preceding the call statement. In particular, to integrate symbolic execution with our system, we are thinking to use JavaPathFinder [APV07]. JavaPathFinder is a tool than can perform program execution with symbolic values. Moreover, JavaPathFinder can mix concrete and symbolic execution, or switch between them. JavaPathFinder has been used for finding counterexamples to safety properties and for test input generation.

To sum up, the main goal of the caller-based slicing algorithm is to facilitate the use of annotated components by discovering statements that are critical for the satisfaction of the precondition, i.e., that do not verify the precondition or whose statements values can lead to its non-satisfaction (a kind of *tracing call analysis of annotated procedures*).

4.5   Summary

In this chapter we have presented some concepts like Static Slicing, Dynamic Slicing, System Dependence Graph and Procedure Dependence Graph. We have shown the formal definition for the PDG and for the SDG. We have also made the distinction between Static Slicing and Dynamic Slicing. Some approaches applying these concepts were presented and discussed.

Also we have shown a few applications of program slicing on many aspects of the software life cycle. At the end of the chapter we have presented the formal definition of Caller-based Slicing. This is the slicing approach that will be implemented by our tool.

Chapter 5

Tool Proposal

In this chapter, we introduce GamaPolarSlicer a tool that we are building to implement our ideas; it will become available to be used by open source communities, as soon as possible. This project is being developed in the context of the CROSS project - An Infrastructure for Certification and Re-engineering of Open Source Software at Universidade do Minho [1].

First we explain the purpose of the tool and present a few examples to help to understand what we aim to solve with this tool. After that, we describe the architecture of the tool.

The program listed in Example 1 computes the maximum difference among student ages in a class. This component reuses other two: the annotated component `Min`, defined in Example 2, that returns the lowest of two positive integers; and annotated component `Max`, defined in Example 3, that returns the greatest positive of two integers.

Suppose that we want to study (or analyze) the call to `Min` in the context of `DiffAge` program.

For that purpose, the annotated slicing criterion will be:

$$C_a = ((x \geq 0)\&\&(y \geq 0), Min, \{a[i], min\})$$

With this criterion, the first step consists in a backward slicing process performed taking into account the variables present in V_s(`a[i]` and `min`). Then, using the obtained slices, the detection of contract violations is executed. For that, the precondition is back propagated (using symbolic execution) along the slice to verify if it is preserved after each statement. Observing the slice corresponding to the variable `a[i]` (see Example 4 below), it is evident that it can not be guaranteed that all integer elements are greater than zero on account of the function call on line 3 which return value is impredictable; so a potential precondition violation is detected.

---

[1]More details about this project can be found in
http://wiki.di.uminho.pt/twiki/bin/view/Research/CROSS/WebHome

---

Example 1 DiffAge

```
1: public int DiffAge() {
2:     int min = System.Int32.MaxValue;
3:     int max = System.Int32.MinValue;
4:     int diff;
5:
6:     System.out.print("Number of elements: ");
7:     int num = System.in.read();
8:     int[] a = new int[num];
9:     for(int i=0; i<num; i++) {
10:         a[i] = System.in.read();
11:     }
12:
13:     for(int i=0; i<a.Length; i++) {
14:         max = Max(a[i],max);
15:         min = Min(a[i],min);
16:     }
17:
18:     diff = max - min;
19:     System.out.println("The gap between max and min age is " + diff);
20:     return diff;
21: }
```

Example 2 Min

$/ * @$ requires $x \geq 0$ && $y \geq 0$
$@$ ensures $(x > y)?$ \result $== x :$ \result $== y$
$@ * /$

```
1: public int Min(int x, int y) {
2: int res;
```
3: $res = x - y;$
4: **return** $((res > 0)?$ y : x);
```
5: }
```

Example 3 Max

$/ * @$ requires $x \geq 0$ && $y \geq 0$
$@$ ensures $(x > y)?$ \result $== y :$ \result $== x$
$@ * /$

```
1: public int Max(int x, int y) {
2: int res;
```
3: $res = x - y;$
4: **return** $((res > 0)?$ x : y);
```
5: }
```

Example 4 Backward Slice for a[i]

1: $int[]a = newint[num];$
```
2: for(int i=0; i<num; i++) {
```
3: $a[i] = System.in.read();$
```
4: }
5: for(int i=0; i<a.Length; i++) {
```
6: $min = Min(a[i], min);$
```
7: }
```

5.1 Architecture

Being aware of the issues discussed before, the idea is to help guarantee the correct behavior when integrating an annotated component into a new system reusing it, creating a tool to automate this process. This integration should be smooth, in the sense of that it should not turn a correct system into an incorrect one.

To achieve this verification goal, it is necessary:

- to verify the component correctness with respect to its contract (using a traditional Verification Condition Generator, already incorporated in GamaSlicer[23], available at http: //gamaepl.di.uminho.pt/gamaslicer);

- to verify if the actual calling context preserves the precondition;

- to verify if the component is properly used in the actual context after the call;

- Given a reusable component and a set of calling points, specify the component body according to the concrete calling needs.

The whole process is a bit complex and was divided in a set of smaller problems (*divide and conquer*). The tool under discussion in this Master work will only focus on the second item, working with preconditions and backward slicing.

Figure 5.1 shows GamaPolarSlicer architecture, aiming at the easiness of the described process.The architecture is based on the classical structure of a language processor.

Source code can be a Java project or only Java files to analyze by the tool.

Lexical Analyzer, Syntactic Analyzer, Semantic Analyzer the Lexical layer converts the input into symbols that will be later used in the Identifier Table. The Syntactic layer uses the result of the Lexical layer above and analyzes it to identify the syntactic structure of it. The Semantic layer adds the semantic information to the result returned by the Syntactic layer. It is in this layer that the identifier table is built. These three layers, usually are always present in language processors.

Invocations Repository is the data structure where all function calls processed during the code analysis are stored. The contract verification will be applied to each one of these calls and the slicing criterion of each one will consider the parameters struct.

Annotated Components Repository is the data structure where all components with a formal specification (precondition and postcondition at least) are stored. All these components will be later used in the slicing process in order to filter all the calls (from the invocation repository) defined without any type of annotation. This repository has an important role when verifying if the call respects the component contract.
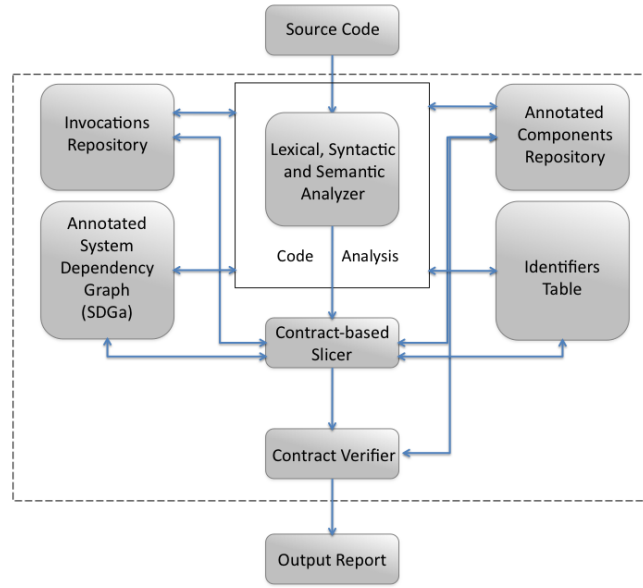
Figure 5.1: GamaPolarSlicer Architecture

Identifier Table flags, always, an important role on the implementation of the processor. All symbols and associated semantic processed during the code analysis phase are stored here. It will be one of the backbones of all structures and of all stages of the tool process.

Annotated System Dependency Graph is the internal representation chosen to support our slicing-based code analysis approach. Constructed during the code analysis, this type of graph allows to associate formal annotations , like preconditions, postconditions or even invariants, to the its nodes. It will be explained in more detail later in this chapter.

Caller-based Slicer is the layer where the backward slicing is applied to each annotated component call. It uses both invocations repository and annotated components repository to extract the parameters to execute the slicing for each invoked annotated component. The resulting slice is a Annotated System Dependency Graph this a subgraph of the original Annotated System Dependency Graph, with all the statements relevant to the particular call.

Contract Verifier using the slice that resulted from the layer above, and using the component contract, this layer analyzes every node on the slice and verifies in all of them if there are guarantees that every annotation in the contract is respected.

Output Report describes all contract violations found during the whole process. All violations found are marked with the degree of relevance in order to aid the user in the revision process. In the future, the tool will the possibility to provide some suggestions to solve these issues, and a graphic display of the violations over the Annotated System Dependency Graph.

5.2 Annotated System Dependency Graph (SDGa)

In this section it is presented the definition of Annotated System Dependency Graph, SDG_a for short, that is the internal representation that supports our slicing-based code analysis approach. To support these definitions, please consider the definitions of PDG and SDG on Chapter 4, Section 4.1.1.

Definition 8 (Annotated System Dependency Graph). *An Annotated System Dependency Graph, SDG_a, is a SDG in which some nodes of its constituent PDGs are annotated nodes.*

Definition 9 (Annotated Node). *Given a PDG for an annotated procedure $\mathcal{P}_a$, an Annotated Node is a pair $< S_i, a >$ where S_i is a statement or predicate (control statement or entry node) in $\mathcal{P}_a$, and a is its annotation: a pre-condition α, a post-condition ω, or an invariant δ.*

The differences between a traditional SDG and an SDG_a are:

- Each procedure dependency graph (PDG) is decorated with a precondition as well as with a postcondition in the entry node;

- The *while* nodes are also decorated with the loop invariant (or true, in case of invariant absence);

- The *call* nodes include the pre- and postcondition of the procedure to be called (or true, in case of absence); these annotations are retrieved from the respective PDG and instantiated as explained below.

To give a better idea of how we can create a SDG_a from a source program, and its importance when working with annotated components, we decided to present a few small examples. We hope these will help to understand why we find the SDG_a important to our work.

Given a program and an annotated slicing criterion, we identify the node of the respective SDG_a that corresponds to the criterion (yellow node in Figure 5.2). After building the respective caller-based slice, the critic statements will be highlighted in the graph, making easier to identify the statements violating the precondition (red nodes in Figure 5.2).

All kind of statements are represented by nodes in the graph. For example, the statement a[] = new int[num] would be a node like:

a[] = new int[num]

The statements can have data flow dependencies between them. These dependencies are the edges of the graph and are represented using a dashed arrow.

Given the following piece of code

```
int min = System.Int32.MaxValue;
int max = System.Int32.MinValue;
diff = max - min;
return diff;
```

the corresponding SDG_a would be

min=System.Int32.MaxValue   max=System.Int32.MinValue   diff=max-min   return diff
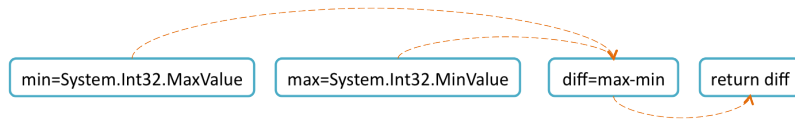
Besides data flow dependencies, the statements can also have control flow dependencies. Control flow dependencies occurs when a statement x needs a statement y to run in order to be able to run too.

The following code is an example of statements with control dependencies

```
public int DiffAge() {
    int min = System.Int32.MaxValue;
    int max = System.Int32.MinValue;
    a[] = new int[num]
    i = 0;
}
```

In the SDG_a we represent control flow dependencies with an arrow as we can see below

Entry DiffAge

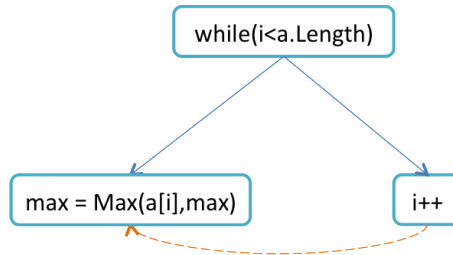min=System.Int32.MaxValue   max=System.Int32.MinValue   a[] = new int[num]   i=0
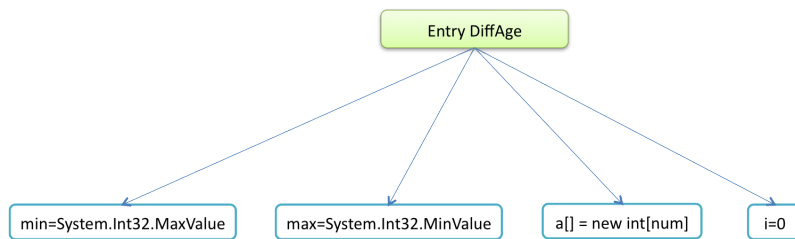
Now we will show how we represent more complex statements, like the conditional statement or the loop statement.

The loop statement works the same way as already seen. For example, the following

while

```
while(i < a.Length) {
    max = Max(a[i],max);
    i++;
}
```

would generate a graph like this

On the other hand, the conditional statement implies a change in the node representation as we can only follow one of the control dependencies edge of it.

For the following conditional statement

```
if(res > 0) {
    res = x;
}
else {
    res = y;
}
```

we would have three nodes. The node for the if would be a diamond. This node also represents the else causing it to have a control dependency to the true condition case and another one to the false condition case, both under the same node as we can see below.
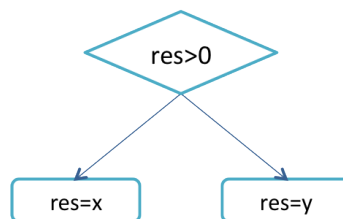

As already seen before, one of the big advantages of this type of SDG is that we can associate formal annotations to the nodes. A node with a precondition and a postcondition would be represented this way:

In a SDG$_a$ we can also represent inter procedural data or flow dependencies. The parameter passing mechanism is made explicit.
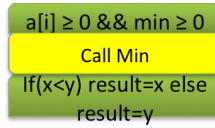
In the graph below we can see the a dependency represented between the parameters on the Min call and the parameters on the entry for the procedure Min. We can also see the dependency between the result of the procedure and the result of the call to it.
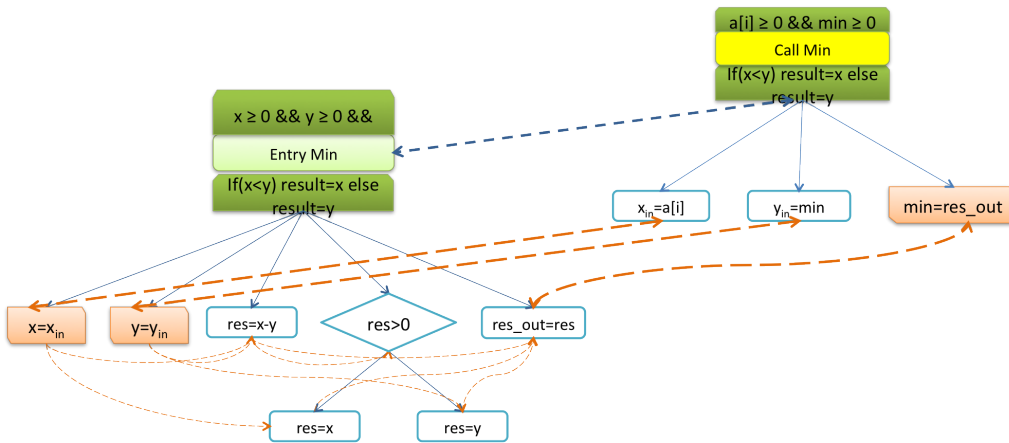
Applying the schemas seen for all the components, we are able to construct a graph from a given piece of source code. Below is the SDG_a for the illustrative example presented in this chapter (Examples 1 to 3). The nodes that are responsible for a possible violation are also marked on the graph.

5.3   Summary

In this chapter we have presented GamaPolarSlicer and explained the context in which the idea for its development emerged. We have presented the tool architecture and described in detail each of the components that assemble it. We have also show the formal definition of the dependence graph representation of a program that supports our slicing-based code analysis approach (the SDG_a). We have explained how the different types of statements are represented as nodes in the SDG_a.
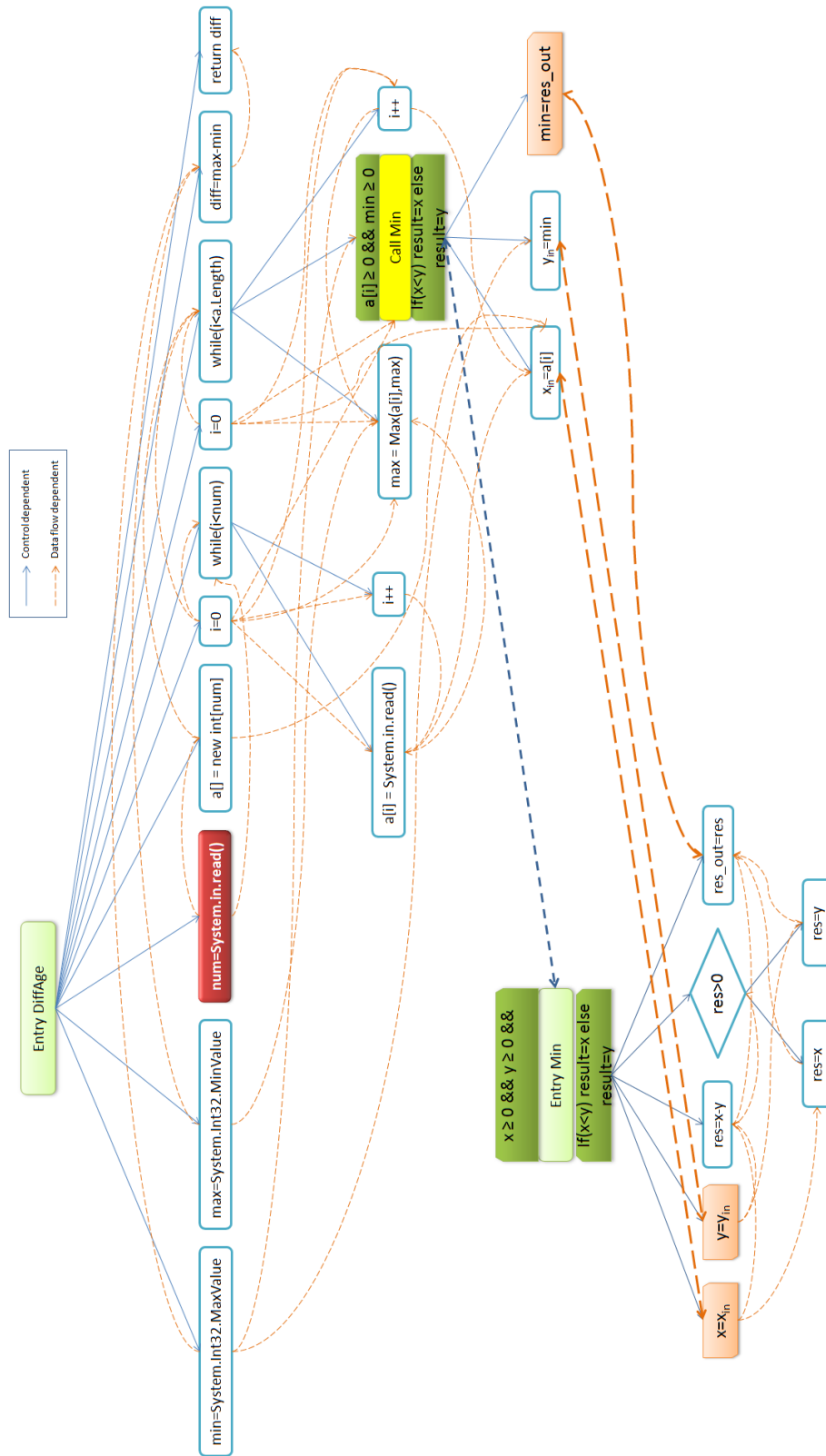
Figure 5.2: SDG_a for a program and its role on Caller-based Slicing

Chapter 6

Tool Implementation

To address all the ideas, approaches and techniques presented in this proposal, it was necessary to choose the most suitable technologies and environments to support the development.

To address the *design-by-contract* approach we decide to use the Java Modeling Language (JML [1]). JML is a formal behavior interface specification language, based on design-by-contract paradigm, that allows code annotations in Java programs [LC03].

JML is quite useful as allows to describe how the code should behave when running it [LC03]. Preconditions, postconditions and invariants are examples of formal specifications that JML provides.

As the goal of the tool is not to create a development environment but to enhance an existing one, we decided to implement it as an Eclipse [2] plugin.

The major reasons that led to this decision were: the large community and the existing support. Eclipse is one of the most popular frameworks to develop Java applications and thus a perfect tool to test our goal; it includes a great environment to develop new plugins. The Plugin Development Environment (PDE [3]) allows a fast and intuitive way to develop Eclipse plugins; it has a built-in support for JML, freeing us from checking the validity of such annotations.

After the first days of the development process we realized that Java has a limitation regarding the number of bytes per class (only allows 65535 bytes per class). This limitation prevented us of continue the work with Java because the parser we were generating for Java/JML grammar exceeded this limit of bytes. This led us to abandon the idea of the Eclipse plugin and implement GamaPolarSlicer using Windows Forms and C# (under .NET framework).

---

[1]http://www.cs.ucf.edu/ leavens/JML/
[2]http://www.eclipse.org/
[3]http://www.eclipse.org/pde/

6.1   Tool Workflow

As depicted in the architecture (see Figure 5.1), our tool is divided in a set of phases where each one solves a particular task. In this section we will explain how these phases interact with each other and how data flows between them.

The tool begins analyzing the source code (Java code/JML annotations) in order to extract all symbols and to construct all data structures. In order to ease the slicing process it is mandatory to have an appropriate data structure to support this type of techniques. For this job we have chosen the Annotated System Dependency Graph(SDG_a) has previously said (see Chapter 5). Using all the gathered information during the code analysis we are able to construct this graph.

The graph and the Identifier Table construction are made once for each input file processed. At the end of these steps, the system will have a set of Identifier Tables and a set of SDG_a. The union between all the SDG_a will result in the SDG_a for the entire source code. The same happens to the set of Identifier Table.

After building all the data structures, the backward slicing is then applied to a component invocation and the resulting slices together with the component contract are used to verify if its call respects the contract. These steps are applied to the set of calls resulting of the intersection between the Invocation Repository and the Annotated Components Repository.

During this process (depicted in the Figure 6.1), if a violation is found, a textual report is issued. Also a graphic report can be selected. This graphic report uses the constructed SDG_a.

6.2   Parser

In order to improve the readability and the efficiency of the code, we decided to use an EBNF(Extended Backus-Naur Form) grammar coded with AnTLR to generate a recognizer to the Java language. During this section we will show how we constructed the various data structures and how we processed the annotations found on the code.

AnTLR is a tool released in 1992 by Terence Parr [4]. AnTLR is a parser generator that uses LL(*) parsing.

The job of AnTLR in this work is to generate a recognizer for the Java language to be later used by the tool to parse the input and construct all the data structures. This recognizer is generated in the C# language to ease the integration of it later on.

The grammar used as input for the AnTLR was firstly developed by Terence Parr and later updated by AnTLR community.

This grammar specifies the Java language with the addition of JML annotations.

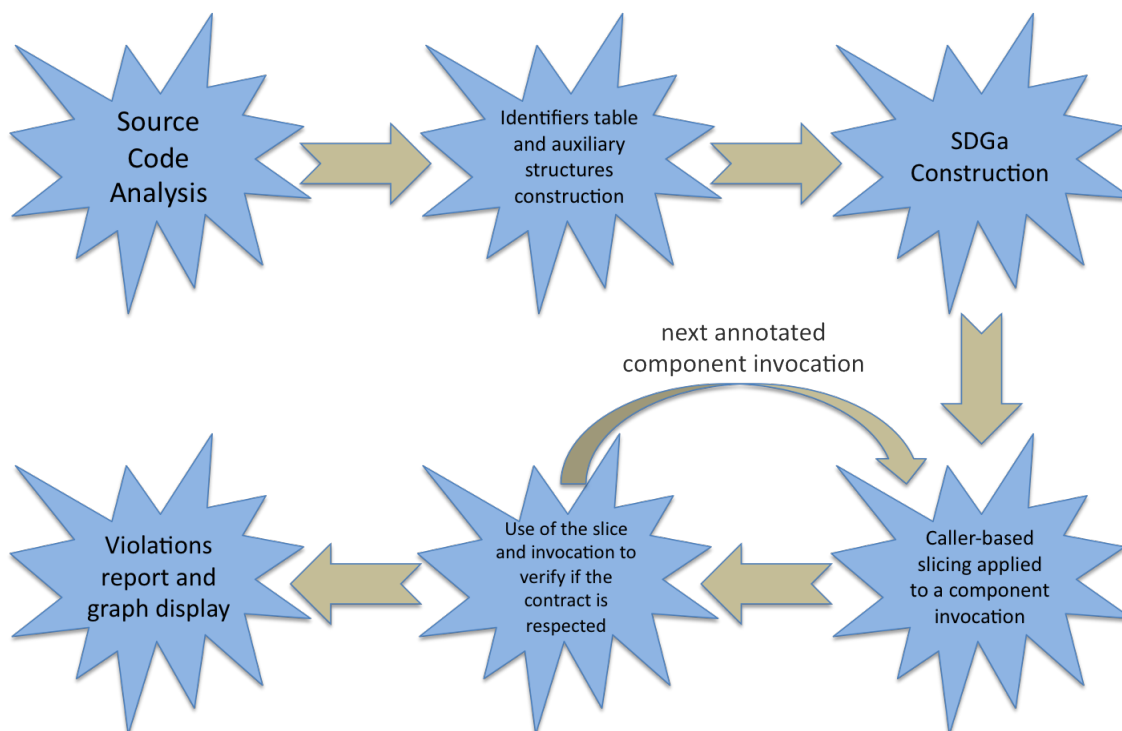[4]http://www.antlr.org/wiki/display/ admin/Home

Figure 6.1: Tool Workflow

This improvement was made by Daniela da Cruz [5].

Our contribute to this grammar focused on the semantic actions. Actions to construct the Identifiers Table, the repositories and the SDGa were implemented.

6.2.1 Identifier Table

As usual in this type of work, the Identifier Table is always present and it is used by the tool as knowledge support. All the relevant information concerning the processed symbols is stored here.

The Identifier Table is divided in four different C# classes: `Entry`, `Scope`, `Table` and `MyType`.

The `MyType`, as the name implies, represents the type of a symbol. The table works in a similar way as multiple linked lists distinguished by their scope. Each recognized symbol is treated as an entry, and each entry has a scope associated to it. Each block of statements has a different scope and it is represented with a new linked list hanging on the entry to that block. This would not be necessary if we did not want to store all the symbols in a scope until the program end. Figure 6.2 illustrates how the table is filled.

Below we will show how we add a class or a method to the table in order to better

---

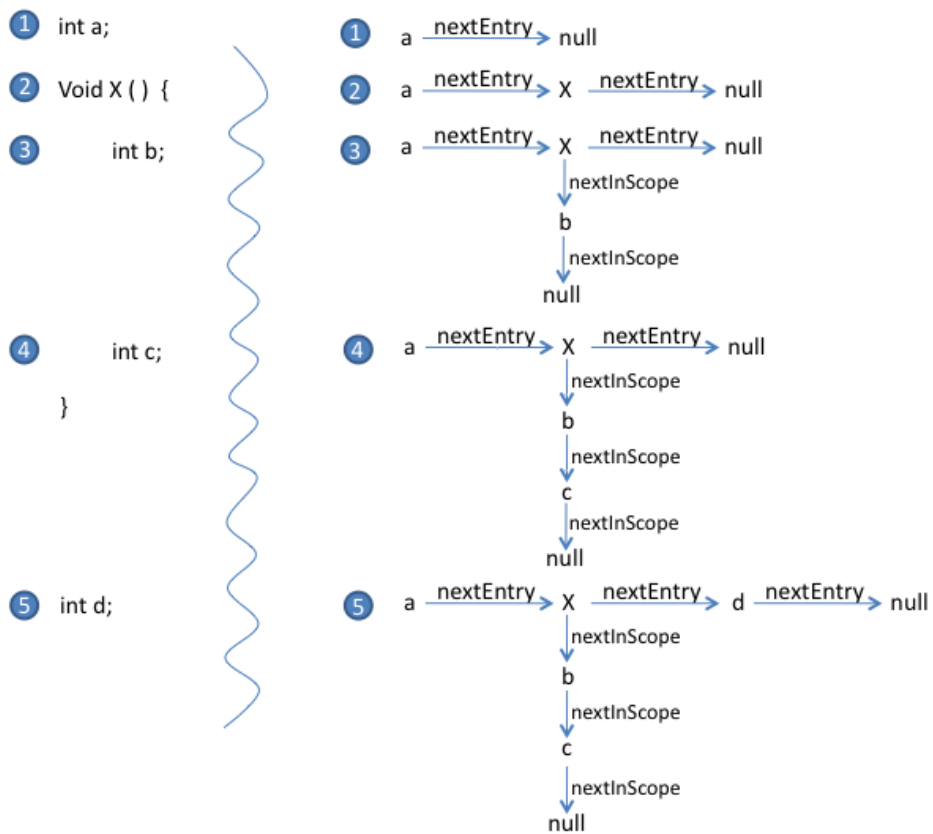[5]http://alfa.di.uminho.pt/ danieladacruz/

Figure 6.2: Identifier Table built in memory

understand how it works.

```
1  'class'  Identifier  (typeParameters)?      {
2         classEntry = $symTab.insert(Entry.Typ, $Identifier.text, new
            MyType(MyType.Class, MyType.voidType, $Identifier.text),
            $isPublic, $isStatic, $isFinal, owner);
```
Listing 6.1: Add an entry to the table

To a class, first we extract the name of it using the terminal symbol IDENTIFIER. With this information we create an entry in the table for this class. After that and before processing the symbols in the class body, we open a new scope in the table for the class (as seen before, each class has a different Identifier Table).

```
1  ('extends' type)?
2  ('implements' typeList)?
3         { $symTab.openScope(0); }
```
Listing 6.2: Open a new scope

Now we just need to make sure that the grammar production that processes the class body inherits the class owner, i.e. , the owner of the following statements of the code (classEntry).

```
1  outer=classBody[symTab, classEntry, inGraph]
```
Listing 6.3: Inheritance of the owner class by the production for the class body

In the case of a method, we begin to identify its modifier. As we can have different ways to declare a method, we also have different productions on the grammar to recognize them. We will present here only one, as there are only a slightly changes between them.

After the modifiers, we have the return type of the method. This is another information that the production to recognize the method declaration, must include as inherit attributes.

```
1  :    type!        { outType = $type.outType; $inMethod.ret.type = $type.
    outType.name; }
2         out1=methodDeclaration[isPublic, isStatic, isFinal, outType,
            symTab, owner, inMethod]
```
Listing 6.4: Method return type

Now we just need to recognize the method name in order to have all the necessary info to create an entry on the table for this method.

```
1  :    Identifier { name = $Identifier.text;
2         method = $symTab.insert(Entry.Mth,$Identifier.text,type,
            $isPublic,$isStatic,$isFinal,owner);
```
Listing 6.5: Create a table entry for a method

With the entry created, the rest of the process is similar to what we already seen when adding a class; We open a new scope and put the entry as an inherit attribute in the production that recognizes the method body.

At the end, the table will have all the necessary info from all symbols recognized during the input analysis, and this information will remain stored until the end of processing.

6.2.2   Repositories

As the analysis and the contract verification are not simultaneous phases, the role of both, Annotated Components Repository and Invocations Repository, is crucial in the development process.

The intersection between these repositories give us the set of calls to which the contract verification will be applied. Also, during the contract verification the Annotated Components Repository is used to provide the precondition and postcondition in cases of invocations found in the slicing result. We will explain it in more detail later in the document (see section 6.3).

To add a new call to the Invocation Repository, involves merely a verification to the number of arguments of the call. If an invocation has no arguments then is discarded since does not follow the purpose of this work.

```
1 |    a=Identifier ('.' e=Identifier)* (c=identifierSuffix[idTree])?
2 {
3     if(null != $c.outArgsList) {
4         Call node = new Call($a.text + auxcode,"",$a.line ,"","");
5             node.owner = call_owner;
6           node.call_line = call_line;
7           invocationsList.Add(node);
8     }
```

Listing 6.6: Add a new call to the Invocation Repository

To add a new annotated component to the Annotated Components Repository is very similar. There is just the need to verify if the recognized method also includes a JML specification, otherwise it is discarded. It is only required the existence of a pre condition in the JML specification. The existence of a postcondition is not mandatory in the context of GamaPolarSlicer.

```
1 |  (a=jmlMethodSpecification?) modifiers
2 outerM=memberDecl[$modifiers.isPublic, $modifiers.isStatic, $modifiers.
      isFinal, symTab, owner, inMethod]
3 {
4     if ($outerM.isMethod)
5     {
6         $numMethods = 1;
7         if(null != $outerM.outMethod)
8         {
```

```
9              if ( null != a) {
10                  $outerM.outMethod.precondition = $a.pre;
11                  $outerM.outMethod.poscondition = $a.post;
12                  $outerM.outMethod.isAnnotated = true;
13                  annotatedComponents.Add($outerM.outMethod);
14              }
```

<div align="center">Listing 6.7: Add a new annotated component to the Repository</div>

6.2.3   Annotated System Dependency Graph

The construction of this structure is much more complex than others seen before. The decision about the structure to use fell on the adjacency list due to the flexibility it allows at the time of its implementation. Our implementation differs a bit from the usual as the nodes are not all as entries on the list, instead we only have entries for the nodes representing methods.

As the code can have methods with the same name (polymorphism), we decided to use the line to create the key to the node in the list. For example, if we have two methods with the name `Sum`, one in the line six, and the other on the line twelve, the key to the first node would be `Sum:6`, and to the second `Sum:12`.

As in a SDG_a we can have annotations associated to nodes, then the node for a method must store the precondition and the postcondition, in the case of have them.

Each method node has a list of all nodes connected with a control dependency. This type of nodes are a bit different of the one for the methods. These nodes take advantage of Object-Oriented inheritance. All type of statements are represented as `Node` (abstract class) and when needed they are converted to their type of nodes: `Loop`, `Conditional`, `Call`, `Assignment`, etc..

Nodes that represent a block of code, like a `Loop` node or a `Conditional` node, include a list that contains all the statements on the block which in turn are also of type `Node`.

Figure 6.3 illustrates how the graph is created in memory from the source code.

Until this point we have presented how the graph was implemented. Now we will show how we built it with a grammatical specification.

Most of the updates on the graph occur at the same time that occurs for the Identifier Table, and the implementation is quite similar with a few exceptions that we will explain with more detail.

Assignment or Variable Declaration

Assuming the grammar:

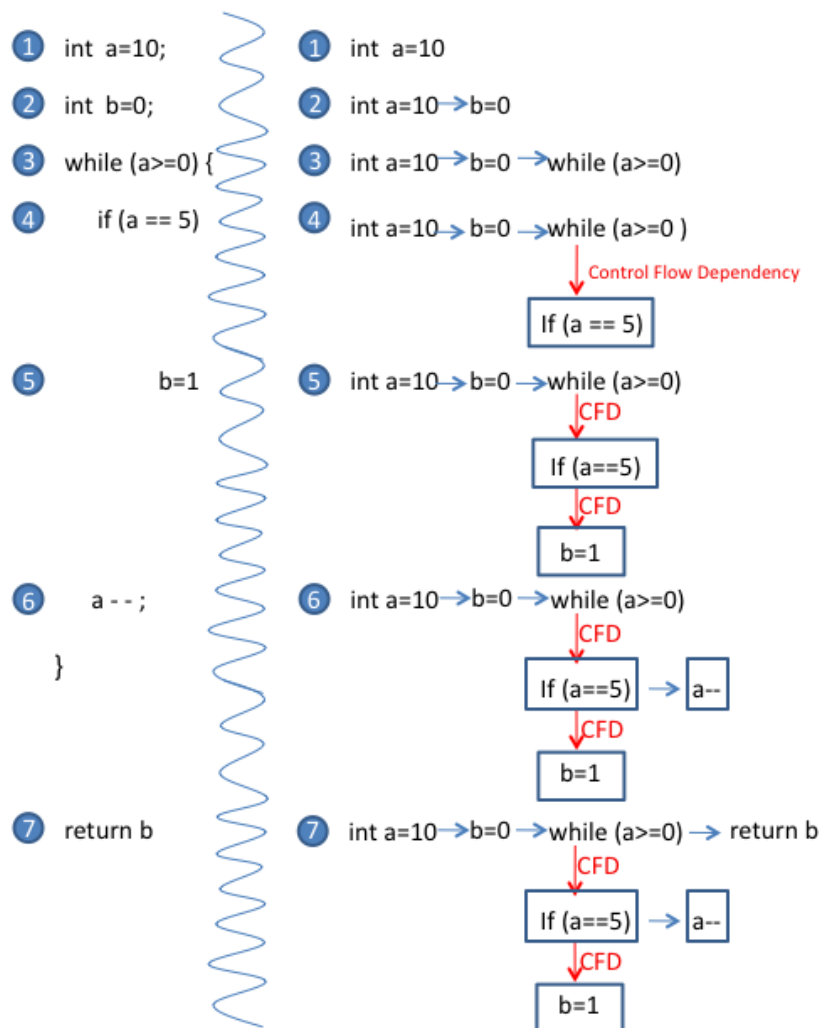<div align="center">Z----->Y   Z</div>

Figure 6.3: Graph built in memory

```
Y----->X    Y
X----->A    X
A----->a
```

To reach the symbol X, the parser will have always to check the productions Z and Y. That is what happens in our grammar with the assignment and calls. The way the grammar was developed, forces the recognizer to check the productions for an assignment or for a variable declaration every time a call is being processed.

As the call and the assignment are different types of nodes and have a particular data type, then this bring us a little inconvenient.

When we have a statement that is a call or is an assignment with no calls on its right side, then there is no problem. The problem is when we have an assignment with a call on the right side. When that happens, we want to create a node of the type `Call` instead of a node of `Assignment type`. The left side of the assignment is stored in the `Call` node as the destination symbol of the call result.

To do this, we begin to increment the number of possible assignments every time the production for assignments (`AssignExpr`) is checked. We do this to avoid the creation of multiple assignments node for the same statement. As we are just interested to have the leftmost assignment of the statement, and using the tree of the `AssignExpr` production we can reach it again, then this counter will give us the guarantee that we always know who is the leftmost assignment.

```
1 assignExpr returns [string value, string outCode, int line, Node
    outNode]
2      :     {isAssign++;} c=conditionalExpression (a=assignmentOperator
            b=assignExpr)?
```

Listing 6.8: assignExpr production

When we find the leftmost assignment there is the need to verify if the right side is a call or not. If it is a call we ignore the assignment node as we already have a node attribute to synthesize. If it is not, we have to create an `Assignment` node to assign to the attribute node to synthesize.

```
1 if(1 == isAssign) {
2     if($b.outNode != null) {
3         ((Call)$b.outNode).ret.var = $c.outCode;
4         $b.outNode.codeline = $c.outCode + $a.outCode + $b.outCode;
5         $outNode = $b.outNode;
6     }
7     else {
8         $outNode = new Assignment($c.outCode,$c.outCode + $a.outCode +
            $b.outCode,$c.line,$c.outCode,$b.outCode,$a.outSymbol);
9     }
10 }
```

Listing 6.9: Assignment nodes

But we can have more expressions beside calls and assignments. All those expressions are considered of the type `Expression`. These nodes are created every time the `AssignExpr` has the synthesized attribute node empty.

```
1 expression returns [string value, Node outNode, String outCode, int
      line]
2     :    a=assignExpr {
3             if(null == $a.outNode) {
4                 $outNode = new Expression($a.outCode,$a.outCode,$a.line
                    ,$a.outCode);
5             }
6             else {
7                 $outNode = $a.outNode;
8             }}
9        ;
```

<div align="center">Listing 6.10: Expression nodes</div>

IF/ELSE IF

In practical terms the `if/else if` structure is very similar to the case structure. Multiple comparisons and the default case that can be taken as the final `else`.

Again, the way the grammar was designed brought us a few drawbacks. In the grammar there is no distinction between an `if/else` and an `if/else if` statement. We had to find a way to distinguish both statements in order to create the right node. In the graph, the node for an `if/else if` statement is represented in the same way as the node for `switch` statement.

Every time the production for and `if` statement is called, it is made a verification to check if this `if` is after an `else` making it an `else if` (`else if flag`). Also, we count how many times we do this verification in order to distinguish every time the production is used for an `if` or for an `else if`.

```
1 |    'if' {
2         if(true == isParentIf) {
3             isElseIf = true;
4         }
5         isTheIf++;
6     }
```

<div align="center">Listing 6.11: If or Else If verification</div>

After that we must check the `if` body and the `else` statement, in case of it. Here there are a few things that need to be processed and verified as we have to decide between a `Switch` node and a `Conditional` node. After the recognizing of the `if` body, we verify if there is an `else` after it. If the `else` is defined and is the final `else` (`else flag`) we prepare all the `else` information and add it to the list of all `else/else if` statements found consecutively.

```
1 parExpression e=statement[symTab,owner] (options {k=1;}:t='else' {
      isParentIf = true;} f=statement[auxTab,auxOwner])?
2 if(true == $f.isElseOnly) {
3     Condition else_cond = new Condition("_DEFAULT",$t.line);
4     if(null == $f.outNodeList && null != $f.outNode) {
5         else_cond.addCaseLine($f.outNode);
6     }
7     else if(null != $f.outNodeList) {
8         else_cond.casecode = $f.outNodeList;
9     }
10    elseIfCode.Add(else_cond);
11 }
```

<div align="center">Listing 6.12: Final else verification</div>

If the `else if flag` is activated and we are in the presence of the `if` statement, then we must create a `Switch` node.

```
1 if(true == isElseIf) {
2     if(1 == isTheIf) {
3         elseIfCode.Reverse();
4         $outNode = new Switch($parExpression.text,"",$parExpression.
             line,"_IF",elseIfCode);
5         elseIfCode = new ArrayList();
6     }
7 }
```

<div align="center">Listing 6.13: Else if node construction</div>

If the `else if flag` is not activated we just need to check if we have to create a `Conditional` node with or without `else`. The `else flag` is now activated to alert for the end of the `if` statement.

```
1 if(null != f) {
2     $outNode = new Conditional($parExpression.text,"",$parExpression.
         line,$parExpression.text,auxcode,auxcode2);
3 }
4 else {
5     $outNode = new Conditional($parExpression.text,"",$parExpression.
         line,$parExpression.text,auxcode,new ArrayList());
6 }
7 $isElseOnly = true;
```

<div align="center">Listing 6.14: Conditional node</div>

At the end of all these verifications we must decrease the counter, otherwise all `else if` statements will be ignored during the parser.

```
1 isTheIf--;
```

<div align="center">Listing 6.15: Decrease number of if/else if statements found</div>

6.3 Contract Verification Strategies

As already shown, the contract verification is applied upon the slices that result from the caller-based slicing process. This implies the verification of all statements on the slices to check possible verifications. Depending on the statement type, there are a few critical verifications that need to be made. For readable purposes, we will use the following notation in the remainder of this chapter:

- **Call** refers to the function invocation for which we want to apply the contract verification;

- **Caller** is the component where the call occurs;

- **Callee** is the component invoked.

Please consider the example 5 with two annotated components, where one of the components invokes the other.

Example 5 Precondition violation

```
 1: /*@ behavior
 2: @ requires a > 0;
 3: @ ensures pot = a^b;
 4: @ */
 5: public int sqr(int a, int b) {
 6:     int pot = 1, i;
 7:     for(i=0;i<b;i++) {
 8:         pot = mult(a, pot);
 9:     }
10:     return pot;
11: }
12: /*@ behavior
13: @ requires c > 10 && d > 0;
14: @ ensures pot = c * d;
15: @ */
16: public int mult(int c, int d) {
17:     int res = c * d;
18:     return res;
19:     }
```

On the notes in red, we can see that one of the parameters of the call we want to verify is also a parameter on the caller. As the verification is only made on caller (as standalone component), there is no way to verify the value of the parameter at the beginning. This lead us to the first critical verification, precondition versus precondition.

6.3.1 Precondition vs Precondition

When the call and the caller share a parameter we decided to certify it value using the caller precondition. Doing this, we have three possible cases:

1. the caller has an annotation for the parameter and the callee does not;

2. the caller does not have an annotation for the parameter and the callee does;

3. both, the caller and the callee, have an annotation for the parameter.

In the first case, it is obvious that does not change anything. If the callee does not have an annotation for the parameter then it means the parameter can assume any value.

The second case brings ambiguity to the problem. If the caller does not have an annotation for the parameter, then there is no way to guarantee that its value will respect the clause on the call contract. Even if after the verification of all statements, the value respects the clause, that value will always be dependent of the value received as parameter on the caller.

The third case, and the most complex one, gives us chance to predict a value for the parameter on the call moment. With the annotation we can calculate or predict the set of values the parameter can take during the execution of the method. To do this we have created an object with a set of flags that tell us what type of value we have and the range of values that can take.

Please consider that we have the following annotation:

```
requires x>0 && x<200
```

After processing this annotation, the object will have the flags for values higher than, lower than and between activated. The between flag is activated when the annotation contains a closed interval.

These flags also help us to make comparisons between annotations. We can compare preconditions with preconditions and even preconditions with postconditions. The last one is very important to the second critical verification.

6.3.2 Precondition vs Postcondition

Most of all pieces of source code have function calls. When the call of these functions affects the value of a parameter on the call that we are trying to verify, then forces the verification of their postcondition (if defined). This is what we will discuss in this section.

When we found a statement with a function call in the slice result, we verify if the invoked component exists on the loaded source code. If it is an external component, like one included from an imported library, then we have no way to guarantee that the program will work correctly after this point.

During the review process of one of our papers we received a question that raised questions for another issue. The question was, "(. . .) depends on the (human) reader's knowledge that an input function might not have return a positive integer (or even any number); but how does the slicer knows this?" (the given example was using integers). When we identify a call to an external function, we add an entry on the output report with a warning, alerting to the fact that a few verifications must be make in order to guarantee that all calls, to an annotated component, that receive value as parameter, will have the contract respected. We recognize this type of functions using all the data structures constructed during the analysis process. If a call is found in a slicing result, but has no entry on the identifier table, then is considered a call to an external function. Line 10 of the example 1 (chapter 5) is an example of a call to an `I/O` function, and possible contract violation.

Everything discussed until now in this section happen when found a call to an external function. But how about, when the function is on the identifier table and on the repositories? When this happen we have three possible cases:

1. the call we are verifying the contract has no annotation for the parameter with the resulting value of the function call;

2. the found call has no postcondition and the call we are verifying has an annotation for the parameter with the resulting value of the function call;

3. the found call has postcondition and the call we are verifying has an annotation for the parameter with the resulting value of the function call;

In the first case, the result of the found call makes no difference as the parameter has no restrictions of value.

The second case will generate a warning message as we are not able to predict the values of the parameter making impossible to guarantee that the contract will be respected.

The last case force the calculation of the possible values, to be used on the next iterations, using the postcondition. All the information is stored in the objects already seen. These objects are later used to compare the postcondition and precondition annotations regarding a particular parameter in order to find contract violations.

6.3.3   Values vs Precondition

This last critical verification occurs every time during of the verification of the statements on the slicing result. Each time the parameter suffers a change, the values it can take must be recalculated. This may look easier than it really is.

If we have an assignment it is pretty easy to calculate the new value but if we have the same assignment inside an `if` block, for example, the complexity increases significantly. We must assure that both values (if the condition is true and if it is not) are used to compare with the call precondition.

Having all this in consideration, we decided to use a flexible list in order to store the list of values the parameter can accept. Every time we found a new path in the code to reach the call we are verifying, we create a new entry on the list with the calculated value. The way we have defined the object, seen in section 6.3.1, also allow us to compare values with annotations.

In case of violations, these comparisons always lead to error messages. At this point we are able to find contract violations without any doubts so there is no reason to generate warning messages.

6.4 Graphical Interface

The interface was developed aiming at easing the readability and to give a better understanding and visualization of the contract violations in a project. The tool was developed resorting to Windows Forms. Figure 6.4 shows the GamaPolarSlicer graphical interface.

The interface is divided into three windows: two small windows located on the left and one main window on the right, with four tabs in it.
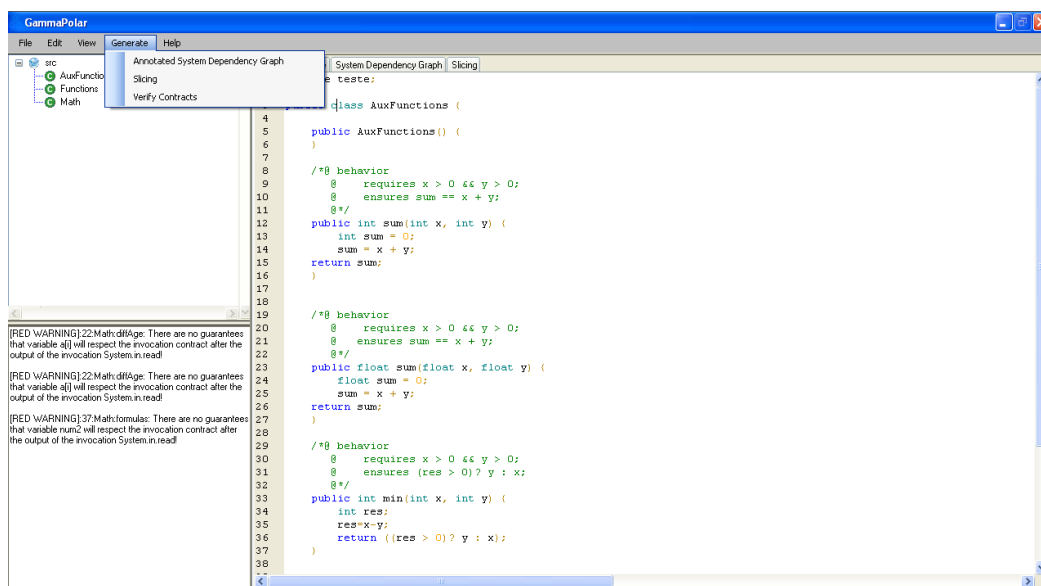
Figure 6.4: GamaPolarSlicer Graphical Interface

The tool provides a tree view to show all the components of the loaded project, and a text box to present the violations found during the verification. It also provides four type of views: the Code View, the Identifier Table View, the SDG_a View and the Slicing View.

All these tool components will be presented in more detail in the remainder of this section.

The tool provides an easy way to navigate within the project. A tree view, displayed on the top left window, is available with all the classes, packages or folders arranged hierarchically as we can see in Figure 6.5.
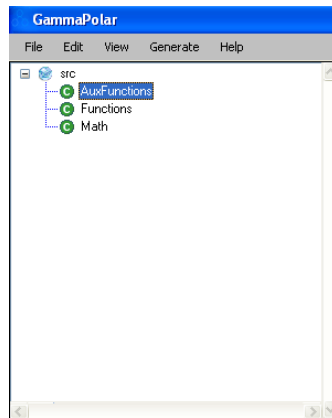

Figure 6.5: GamaPolarSlicer Tree view of the project content

If a double click is issued on a tree item, the info (Code, Identifier Table, etc.) of the selected class will be loaded and presented on the four tabs of the main window on the right side. Not all the info will be loaded as the slicing information or the SDG_a needs the user request.
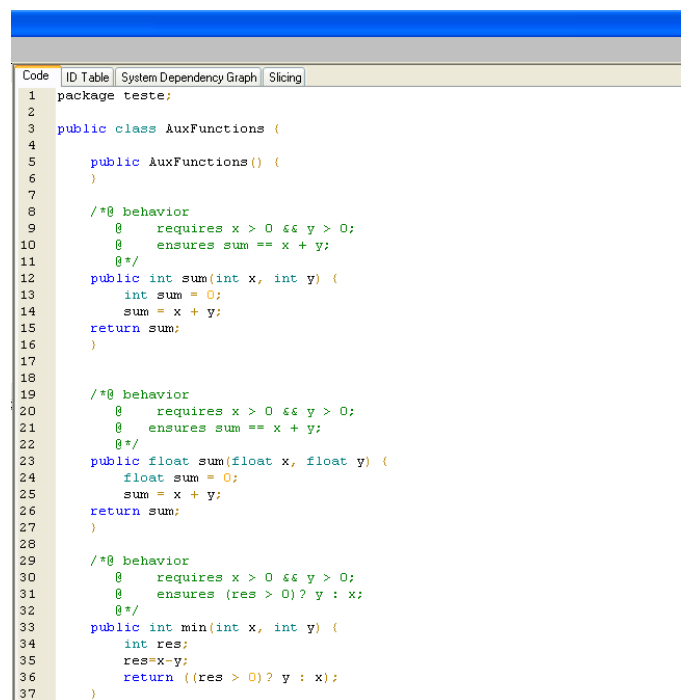
Figure 6.6: GamaPolarSlicer Code view

By default (after loading a software system and before selecting a tree item) the main window displays the information related to the first item of the tree.

The user has access to the Code view as the default view (the first tab on the right main window). The Java code is highlighted to increase its readability. To do this we used scintillaNet [6]. This is a library that can be imported by Visual Studio, that provides us a special text box where we can define all the color definitions we want to highlight our code.

Figure 6.6 shows how a code fragment looks like when imported to our tool.

The Identifier Table view shows the information collected for all symbols in the selected class. The information can be filtered in order to visualize only the details of the symbols in a particular method selected by the user.

Figure 6.7 shows the identifier table of an entire class on the top, and the information of a single method after filtering on the bottom.

Figure 6.7: GamaPolarSlicer Identifier Table view of a class and a method

Finally when the user requests a verification of the components contracts, if one or more violations are detected the error message will be displayed in the left bottom window; the tool will present a textual description for each violation, marking their position on the source code (Figure 6.8). In the future, we intend to highlight all violations on the source code (first tab), and highlight them also on the SDG_a diagram (third tab).

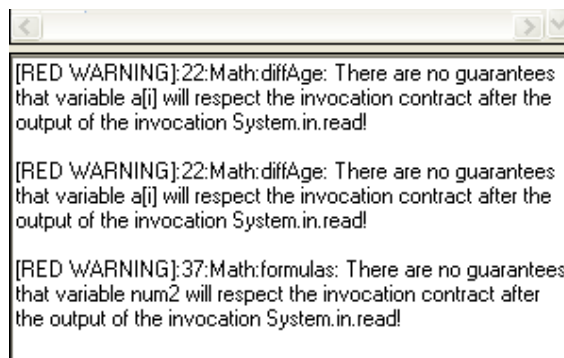
[6]http://scintillanet.codeplex.com/

Figure 6.8: Contract violations alert

It is almost certain that the interface will suffer a few more changes in the future, besides those already mentioned, to be as similar as possible with the others already developed within our group.

6.5   Summary

In this chapter we started by explaining how every tool component interacts with each other and how data flows between them. We have also shown how we constructed all data structures that will support the tool. We have discussed the variety of algorithms we implemented and how we use them to verify the components contracts. It was also presented the graphical interface of the tool, and described the views it provides: Code View, Identifier Table View, SDG_a View and Slicing View. At the moment, the last two are not fully finished and only show some information after the user request for a contract verification.

Chapter 7

Case Studies

To test the performance of the tool we have used four case studies. Three of them are packages extracted from online repositories -Linked Queue, Dinning Philosophers, Bounded Buffer-. Most of the components in these packages have JML specifications.

The fourth package -Math Functions- was developed by us with a few JML specifications to test the answer to most of the contract verifications that the tool does.

The reasons that lead us to choose these packages, instead of others, were the complexity and the variety of JML specifications. In this section we will show in detail these packages and the respective results given by the tool.

7.1 Linked Queue

The first package contains the implementation of a linked queue with concurrency. The components in it are not very complex and the JML specifications are formed by a low average number of annotations (see Example 6).

Example 6 Example of annotated methods from the Linked Queue package

```
/*@ behavior
  @    ensures true;
  @*/
public Object peek() {
    synchronized (head) {
        LinkedNode first = head.next;
        if (first != null)
            return first.value;
        else
            return null;
    }
}
```

The package is formed by four classes (see Figure 7.1): `LinkedQueue`, `Process`, `LinkedQueueDriver` and `LinkedNode`. In Figure 7.1 we can see all the symbols in one of the classes.

Figure 7.1: Identifier Table for the class LinkedQueue

One of the important objectives is to see how the tool answers to the contract verification requests.

Figure 7.2: Violations found during the verification of the Linked Queue project

Errors   As can be seen on the Figure 7.2, the report contains a warning because the context of one of the calls does not guarantee that its contract will be respected.

Performance (time)   The time used by the tool to load and parse (create all data structures) the package is close to one second. The contract verification is almost immediate.

7.2   Dinning Philosophers

The second package contains the implementation of the dinning philosophers concurrency problem, stated by Edsger Dijkstra. This problem is often used to illustrate the problem of deadlock in a system.

The dinning philosophers consists in a circular table where philosophers can only be eating or thinking, and these actions can only be done one at a time. Each philosopher has necessarily a fork on his left and another on his right. To eat, they need to have both forks in their hands.

Compared to the first case study, the components in this one are very similar regarding the complexity, but the JML specifications have a significant increase on the average number of annotations and on their complexity (see Example 7).
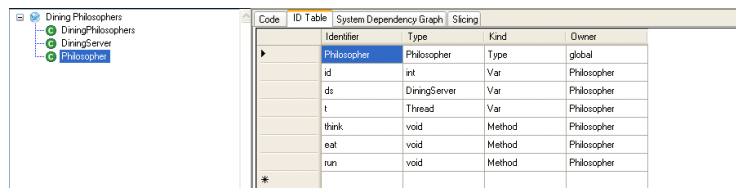

Example 7 Example of annotated methods from the Dinning Philosophers package

```
/*@ behavior
 @    assignable this.numPhils, this.checkStarving;
 @    ensures this.numPhils == numPhils &&
 @            this.checkStarving == checkStarving &&
 @            \fresh(state) && state.length == numPhils;
 @*/
public DiningServer(int numPhils, boolean checkStarving)
{
    this.numPhils = numPhils;
    this.checkStarving = checkStarving;
    state = new int[numPhils];
    for (int i = 0; i < numPhils; i++)
    {
        state[i] = THINKING;
    }
}
```

This package is formed by three classes (see Figure 7.3): `DinningPhilosophers`, `DinningServer` and `Philosopher`. In Figure 7.3 we can see the Identifier Table of one of these classes.

Figure 7.3: Identifier Table for the class Philosophers

Errors   No violations found and immediate answer by the tool.

Performance (time)   Concerning the time used by the tool to load and parse the whole package, compared with the first case study, it needs nearly one more second to do it. The contract verification is immediate. This due to the fact that no violations were found during it.

7.3 Bounded Buffer

The third package contains the implementation of the well-known bounded buffer data structure.

Concerning the components complexity, it increases a bit compared to the two cases studies seen before. Regarding JML specifications, the average number of annotations is lower compared with the second case study, but their complexity is bigger (see Example 8).

Example 8 Example of annotated methods from the Bounded Buffer package

```
/*@ behavior
    @   when count != 0;
    @   assignable buffer[*], takeOut, count;
    @   ensures takeOut >= 0 && takeOut < numSlots &&
    @           \result != null;
    @*/
 public synchronized Object fetch() {
    Object value;
    while (count == 0) {
        try {
            wait();
        }
        catch (InterruptedException e) {}
    }
    value = buffer[takeOut];
    takeOut = (takeOut + 1) % numSlots;
    count--; // wake up the producer
    if (count == (numSlots - 1)) {
        notify();
    }
    return value;
}
```

This package is formed by four classes (see Figure 7.4): `BoundedBuffer`, `Consumer`, `Producer` and `ProducerConsumer`.

Figure 7.4 shows the Identifier Table constructed using the information in one of these classes.

Errors   No violations found and immediate answer by the tool.

Performance (time)   The time used by the tool to load and parse the package is very similar to the time used in the second case study. It also consumes approximately one more second than the first. Relating to the contract verification, occurs

Figure 7.4: Identifier Table for the class BoundedBuffer

the same as in the second case study.

7.4   Math Functions

Last but not least, the fourth package (the one developed by us) contains the implementation of some fairly common mathematical functions. Its components complexity is smaller compared to the other three cases studies. The number of statements per component is bigger, but quiet simple. Regarding JML specifications, they are formed by basic annotations with low complexity (see Example 9).

With this case study, we are more interested to test the performance of the tool when doing the contract verification instead of the performance when loading and parsing the package. Summarizing, compared to the other case studies the complexity is smaller but the number of violations is bigger.

This package consists in three classes (see Figure 7.5): `AuxFunction`, `Functions` and `Math`. Figure 7.5 shows the Identifier Table constructed using the information in one of these classes.

Figure 7.5: Identifier Table for the class AuxFunctions

Errors   In Figure 7.6 we can see the output report provided by the tool with all the violations found during the verification. The type of these violations differs which

Example 9 Example of annotated methods from the Math Functions package

```
public static int diffAge() {
    int min = System.Int32.MaxValue, max = System.Int32.MinValue;
    int num, i, diff;
    int[] a;
    AuxFunctions aux_func;
    System.out.print("Number of elements: ");
    num = System.in.read();
    for(i=0; i<num; i++) {
        a[i] = System.in.read();
    }
    for(i=0; i<a.Length; i++) {
        max = aux_func.max(a[i],max);
        min = aux_func.min(a[i],min);
    }
    diff = max - min;
    System.out.println("The difference between the greatest "
                        + "and the smallest ages is " + diff);
    return diff;
}


/*@ behavior
  @    requires a >= 0;
  @    ensures square = sqrt a;
  @*/
public double sqrt(int a) {
    double square = Math.sqrt(a);
    return square;
}
```

implies different checks, i.e., the tool has used different algorithms to each different type of violation found (see chapter 6 section6.3). Some violations found are due to I/O calls, and other due to conflicts in the preconditions comparison.

Figure 7.6: Violations found during the verification of the Math Functions project

Performance (time)   Even with the increase of violations, the tool responded well and the process continued to be almost immediate.

7.5   Summary

In this chapter, we have shown in detail the case studies we used to test the performance of the tool developed by us.

We realized that there is a relation between the complexity of the components, their annotations and the time the tool needs to load and parse the package.

We also realized that the tool has a good performance when doing the contract verification even when we increase the number of contract violations.

In the future, we intend to define a new case study with both of the two worlds; components and annotations complexity, and large number of violations. This is a difficult task due to the lack of annotated open source packages available.

Chapter 8

Tool Assessment

Due to our lack of time, we could not do the tool assessment. This is also due to the fact that the tool becomes a part of CROSS project, therefore having predicted the completion of the works in the next two months. Anyway, in this section we present how we will proceed with the tool assessment when the time comes to make it.

We have planned to do three studies to examine some factors: the Usability, the Performance/Scalability and the Utility.

- **Usability** because we intend to look into the problems that users may or may not have when using the tool. We also want to know if using the tool is a pleasant experience.

- **Performance and Scalability** are two essential variables in the evaluation of such a tool. We count on studying how the tool behaves toward the various tests.

- **Utility** because we intend to know whether it is easier and/or practical to use our tool to help checking and fixing errors in the reuse of annotated components.

8.1 Performance/Scalability

The tests concerning the performance and scalability would be done by us. First we will start by defining the characteristics of the machine we would use to run the tests. These would be (at least): Pentium IV or equivalent, 46b RAM and 520 GB Disc.

With the machine characteristics set, we will explain now the strategy that we would use to do the tests.

All tests to be applied to the tool would have to contribute something more than the tests previously applied. For example, we could first test the impact of code complexity for the time effort to do the code analysis and contracts verification.

The second would investigate the impact of code size for the same time variables. The third could mix a little of both worlds towards to make a comparison of results. The idea is also increasing gradually the size of the code that we are testing to see if the tool scales.

A problem that we face a priori is that is quite difficult to find repositories with annotated code and becomes even more difficult when looking for packages with thousands of lines of annotated code with the desired characteristics. Most likely we will have to code much of the tests because of the scarcity of these.

At the end of all tests, we intend to present the tables with the results, draw some conclusions and thus strengthen those already obtained from the case studies presented in Chapter 7.

8.2   Usability

To reach conclusions on the tool usability we will use the users' judgment, both experienced users and not experienced users. Even accepting that there are a small number of users profiles/patterns, we propose to do the test with a few dozen of users in order to capture all these profiles/patterns. We will try to have half of users with experience and the other half of ordinary users. We chose to include ordinary users because we believe they find issues that experienced users usually not detect.

The test would be done in a room equipped with several machines with similar hardware in attempt to provide the same experience to all users. The test duration is not yet defined but should be somewhere between 15 and 20 minutes. It will be individual and freely, that is, during this test it will be given the freedom to use the tool and its features in the way that each user find more satisfactory.

At the end of the test, we will draw conclusions about the degree of satisfaction regarding the disposal of tool components and the ease of using it. We will also ask users to for suggest some changes in order to increase the overall usability of the system.

8.3   Utility

To get answers about the usefulness of the application, we intend to do a working session. In this section, we explain how we intend to guide this session.

For this session only, we need a spokesman for managing it, and a machine to make the tests. It will be a group session but all kinds of responses will be considered in the final conclusions. The speaker presents a case study and the tool behavior to its appliance. Then the speaker presents an investigation to realize if users preferred to solve their problem with or without the tool aid. The questions in this survey will be something to think carefully, since all the results and conclusions are based on its quality.

As for the case studies that we will use, several of these characteristics will vary to try to obtain comprehensive results. We will try to swap the code complexity and size, and the complexity and size of the annotations. For example, see how users react with small and not very complex case studies, and with very large and complex case studies. We will see if, in both cases, the users would apply the tool, or not, and why.

8.4 Summary

In this chapter, we have seen in detail our proposal to evaluate the tool usability, the tool performance/scalability and its usefulness. It was impossible for us to do this work within the Master program time; as soon as we can do it under CROSS project, we will follow the guidelines here drown.

Chapter 9

Conclusion

In this dissertation, we have shown the importance of reuse and the need to include it on the systematic software development process (Chapter 2). Integrate the reuse in this process implies a big investment although it can bring several benefits. The reuse, if done in a safe way, reduces the costs and the production time, at the same time increasing the quality and the reliability of the software. It is our belief that to have guarantees that the correct system will remain correct after the integration of a reusable component, it is necessary to resort to specifications associated with components.

We saw in Chapter 3 that Design by Contract, more specifically Reuse by Contract concept, is necessary for a safety reuse. We believe that, without this formal approach, it is wiser to built the system from scratch than to reuse.

We have also seen that Slicing (Chapter 4) is a technique that contributes in many ways to the software life cycle. We have distinguished static from dynamic slicing, and we have shown several approaches based on both. Many approaches to help different areas related with software development and maintenance, were described in order to show the advantages of using slicing. Also was explained in detail how the caller-based slicing concept is defined, and how is used by our tool. A tool was presented (Chapter 5 and Chapter 6) to do the verification of components contract when integrated in a new system. The architecture of the tool is based on the classical structure of a language processor. We have also explained all the design decisions. The slicing is applied on the SDG_a and the computed slices are used to do the contract verification for all annotated components invoked in the system.

The tool is capable to apply the Caller-based Slicing to a program and compute precise slices. Also the computed slices are displayed by the tool to ease the comprehension of the program by the developer, allowing him to focus on the relevant aspects of the program. This tool is also very useful on the program comprehension on its general.

We are glad to conclude that our tool is capable not only to verify the components contracts, but also to do it with efficiency and without harming the comprehension of the program by the developer, based on the positive feedbacks that we received from

different persons, from experienced users to reviewers and conference participants.

Our goal was to check if it was possible to do a contract verification with low computing effort and reasonable precision, and taking into account the obtained results, this was successfully accomplished. In the considered case studies (Chapter 7), the tool presents small response times. Our work still needs more empirical studies so that we can strengthen our conclusions regarding its efficiency and reliability (Chapter 8). Due to the fact that we still did not implement the SDG_a visualization, it was not possible for us to verify if it helps or not to understand the found flaws. This was another goal of this tool.

The tool still presents some limitations in the contract verification process. Expand it in order to process annotations regarding any Java data type does not appear to be an easy job. It is trivial when the annotations specify that values of variables can or cannot be equal to `null`. More than that, it becomes very hard to verify them and the tool becomes less accurate.

The algorithm presented in Section 6.3.3, "value vs precondition", is not yet finished. The computation of the value of a variable usually needs other variables values. One of the barriers is that we considered each component as a standalone component, that is, without any dependence with any other component. Any variable dependent on the component parameters will have its value limited by the annotations in the precondition. If the precondition has not annotations related to the parameter in question, then is almost impossible to compute the variable value with precision. These dependencies between variables also force us, if we want to compute the value of a variable, to compute the values of all its dependencies using. These calculations could compromise the efficiency of application.

Another problem is that `while` construction can have statements that influence the control predicate in any position of its body. As the grammar treats the body as a black box, then in these cases we are not able to compute values that are dependent of a loop computation. This adds some inaccuracy to the system.

As we are talking about annotated components, makes all sense to work also with loop invariants. As future work, we intend to:

- improve the contract verification in order to the algorithm use loop invariants when calculating the values of the variables present in the slicing criterion;

- add new types to be processed in the preconditions, besides numeric types;

- display the SDG_a under consideration and show all the contract violations detected highlighted with different colors over the SDG_a diagram;

- add some new features to the tool, including to do the linkage between the graph and the code so we can always know which statement is related to each graph node;

- do the tool assessment and the scalability tests (they are included in the tool assessment). In case we are not able to find annotated package with enough size to make this tests, then we will have to implement them.

To sum up, we could say that our approach is already in a mature state, partly because of the contribution by the reviewers of our papers related to this work [AdCP10, AdCHP10b, AdCHP10a]. Also, during the presentation of the papers in the related conferences, several comments and compliments were done to the tool, mainly in the sense that it could be a useful contribute to software reuse.

Bibliography

[AdCHP10a] Sérgio Areias, Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Gammapolarslicer, a contract-based tool to help on reuse. In Luis S. Barbosa and Miguel P. Correia, editors, *INForum'10 — Simpósio de Informática (CoRTA2010 track)*, pages 137–148, Braga, Portugal, September 2010. Universidade do Minho.

[AdCHP10b] Sérgio Areias, Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Safe integration of annotated components in open source projects. In Luis Barbosa, Antonio Cerone, and Siraj Shaikh (Guest Eds.), editors, *OpenCert'10 — Fourth International Workshop on Foundations and Techniques for Open Source Software Certification (co-located with SEFM'10)*, Pisa, Italy, Sept 2010. Electronic Communications of the EASST.

[AdCP10] Sérgio Areias, Daniela da Cruz, and Jorge Sousa Pinto. Contract-based slicing helps on safety reuse. In Giulio Antoniol, Keith Gallagher, and Pedro Rangel Henriques, editors, *ICPC '10: Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, pages 62–63, Washington, DC, USA, 2010. IEEE Computer Society.

[Agr94] Hiralal Agrawal. On slicing programs with jump statements. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 302–312, New York, NY, USA, 1994. ACM.

[AH90] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256, New York, NY, USA, 1990. ACM.

[APV07] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. Jpf-se: a symbolic execution extension to java pathfinder. In *TACAS'07: Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, pages 134–138, Berlin, Heidelberg, 2007. Springer-Verlag.

[Bal69] R. M. Balzer. Exdams: extendable debugging and monitoring system. In *AFIPS '69 (Spring): Proceedings of the May 14-16, 1969, spring*

joint computer conference, pages 567–580, New York, NY, USA, 1969. ACM.

[Bal04]   F. Balmas. Displaying dependence graphs: a hierarchical approach. *J. Softw. Maint. Evol.*, 16(3):151–185, 2004.

[BC85]   Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, 1985.

[BdCHP10]   José Bernardo Barros, Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Assertion-based slicing and slice graphs. In *SEFM'10 — 8th IEEE International Conference on Software Engineering and Formal Methods*, Pisa, Italy, Sept 2010.

[BE93]   Jon Beck and David Eichmann. Program and interface slicing for reverse engineering. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 509–518, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[BE94]   Thomas Ball and Stephen G. Eick. Visualizing program slices. In *Visual Languages*, pages 288–295, 1994.

[BH93]   Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *Automated and Algorithmic Debugging*, pages 206–222, 1993.

[BHR95]   David Binkley, Susan Horwitz, and Thomas Reps. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.*, 4(1):3–35, 1995.

[Bin93]   David Binkley. Precise executable interprocedural slices. *ACM Lett. Program. Lang. Syst.*, 2(1-4):31–45, 1993.

[Bin99]   David Binkley. The application of program slicing to regression testing. In *Information and Software Technology Special Issue on Program Slicing*, pages 583–594. Elsevier, 1999.

[BJMvH02]   Don Batory, Clay Johnson, Bob MacDonald, and Dale von Heeder. Achieving extensibility through product-lines and domain-specific languages: a case study. *ACM Trans. Softw. Eng. Methodol.*, 11(2):191–214, 2002.

[BMR+96]   Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* Wiley, Chichester, UK, 1996.

[Car89]   Capt James E. Cardow. Issues on software reuse, 1989.

[CCL98]   Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–608, November 1998. Special issue on program slicing.

[CDLM95]   A. Cimitile, A. De Lucia, and M. Munro. Identifying reusable functions using specification driven program slicing: a case study. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*, page 124, Washington, DC, USA, 1995. IEEE Computer Society.

[CDLM96]   Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance*, 8(3):145–178, 1996.

[CF94]   Jong-Deok Choi and Jeanne Ferrante. Static slicing in the presence of goto statements. *ACM Trans. Program. Lang. Syst.*, 16(4):1097–1113, 1994.

[CFG91]   Michael L. Creech, Dennis F. Freeze, and Martin L. Griss. Using hypertext in selecting reusable software components. In *in Proceedings of Hypertext'91*, pages 25–38, 1991.

[CFR$^+$99] E. M. Clarke, M. Fujita, S. P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages, 1999.

[CH96]   Joseph J. Comuzzi and Johnson M. Hart. Program slicing using weakest preconditions. In *FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*, pages 557–575, London, UK, 1996. Springer-Verlag.

[CLLF94]   Gerardo Canfora, Andrea Di Luccia, Giuseppe Di Lucca, and A. R. Fasolino. Slicing large programs to isolate reusable functions. In *Proceedings of EUROMICRO Conference*, pages 140–147. IEEE CS Press, 1994.

[CLYK01]   I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon. Program slicing based on specification. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 605–609, New York, NY, USA, 2001. ACM.

[Dav93]   T. Davis. The reuse capability model: a basis for improving an organization's reuse capability. In *Software Reusability, 1993. Proceedings Advances in Software Reuse., Selected Papers from the Second International Workshop on*, pages 126–133, Mar 1993.

[DBSB90]   P. T. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard. Lassie—a knowledge-based software information system. In *ICSE '90: Proceedings of the 12th international conference on Software engineering*, pages 249–261, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[dCPH09]   Daniela da Cruz, Jorge S. Pinto, and Pedro R. Henriques. Reuse of annotated components. 2009.

[DF06]   Ewen Denney and Bernd Fischer. A generic annotation inference algorithm for the safety certification of automatically generated code. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 121–130, New York, NY, USA, 2006. ACM.

[DKN01]   Yunbo Deng, Suraj Kothari, and Yogy Namara. Program slice browser. *International Conference on Program Comprehension*, 0:0050, 2001.

[dLFM96]   Andrea de Lucia, Anna Rita Fasolino, and Malcolm Munro. Understanding function behaviors through program slicing. In *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)*, page 9, Washington, DC, USA, 1996. IEEE Computer Society.

[DRL+98]   David L. Detlefs, K. Rustan, M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. In *SRC Research Report 159, Compaq Systems Research Center*, 1998.

[ECGN99]   Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 213–224, New York, NY, USA, 1999. ACM.

[Ern00]   Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, Agosto 2000.

[Fav91]   John Favaro. What price reusability?: a case study. *Ada Lett.*, XI(3):115–124, 1991.

[Fel03]   Yishai A. Feldman. Extreme design by contract. In *XP'03: Proceedings of the 4th international conference on Extreme programming and agile processes in software engineering*, pages 261–270, Berlin, Heidelberg, 2003. Springer-Verlag.

[FF95]   William B. Frakes and Christopher J. Fox. Sixteen questions about software reuse. *Commun. ACM*, 38(6):75–ff., 1995.

[FG90]   W. B. Frakes and P. B. Gandel. Representing reusable software. *Inf. Softw. Technol.*, 32(10):653–664, 1990.

[FHR91]   Gerhard Fischer, Scott Henninger, and David Redmiles. Cognitive tools for locating and comprehending software objects for reuse. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, pages 318–328, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[FI94]   William B. Frakes and Sadahiro Isoda. Success factors of systematic reuse. *IEEE Softw.*, 11(5):14–19, 1994.

[FK05]   W. B. Frakes and Kyo Kang. Software reuse research: Status and future. *Software Engineering, IEEE Transactions on*, 31(7):529–536, 2005.

[FLRL00]   Cormac Flanagan, K. Rustan M. Leino, K. Rustan, and M. Leino. Houdini, an annotation assistant for esc/java, 2000.

[FS97]   Bernd Fischer and Gregor Snelting. Reuse by contract. In *ESEC/FSE-Workshop on Foundations of Component-Based Systems*, pages 91–100, Zürich, 1997.

[FT96]   William Frakes and Carol Terry. Software reuse: Metrics and models, 1996.

[Gal89]   Keith Brian Gallagher. *Using program slicing in software maintenance.* PhD thesis, Catonsville, MD, USA, 1989.

[GL91]   Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.*, 17(8):751–761, 1991.

[GO97]   Keith Gallagher and Liam O'Brien. Reducing visualization complexity using decomposition slices. In *Proc. Software Visualisation Work.*, pages 113–118, Adelaide, Australia, 11–12 Dezembro 1997. Department of Computer Science, Flinders University.

[Gop91]   R. Gopal. Dynamic program slicing based on dependence relations. In *Proceedings of the Software Maintenance'91 Conference*, pages 191–200, Sorrento, Italy, October 1991.

[Gri91]   Martin L. Griss. Software reuse at hewlett-packard. 1991.

[HD94]   M. Harman and S. Danicic. Using program slicing to simplify testing, 1994.

[Hen97]   Scott Henninger. An evolutionary approach to constructing effective software reuse repositories. *ACM Trans. Softw. Eng. Methodol.*, 6(2):111–140, 1997.

[HHP+01]   Mark Harman, Robert Hierons, Ub Ph, Chris Fox, Sebastian Danicic, and John Howroyd. Pre/post conditioned slicing, 2001.

[HPR89]   Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.

[HRB90]   Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs, 1990.

[JC93]   Jun-Jang Jeng and Betty H.C. Cheng. Using formal methods to construct a software component library. In *Lecture Notes in Computer Science*, pages 397–417. Springer-Verlag, 1993.

[JM97]   Jean-Marc Jézéquel and Bertrand Meyer. Design by contract: The lessons of ariane. *Computer*, 30:129–130, 1997.

[JZR91]   J. Jiang, X. Zhou, , and D.J. Robson. Program slicing for c - the problems in implementation. In *Proceedings of Conference on Software Maintenance*, pages 182–190. IEEE CSPress, 1991.

[Kan88]   K. C. Kang. A reuse-based software development methodology. pages 194–196, 1988.

[Kin76]   James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[KL88]   B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[KL90]   Bogdan Korel and Janusz Laski. Dynamic slicing of computer programs. *J. Syst. Softw.*, 13(3):187–195, 1990.

[KR98]   B. Korel and J. Rilling. Program slicing in understanding of large programs. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, page 145, Washington, DC, USA, 1998. IEEE Computer Society.

[Kri04]   Jens Krinke. Visualization of program dependence and slices. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 168–177, Washington, DC, USA, 2004. IEEE Computer Society.

[Kru92]   Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992.

[KtSCH99]   Taeho Kim, Yeong tae Song, Lawrence Chung, and Dung T. Huynh. Software architecture analysis using dynamic slicing. In *AoM/IAoM 17th International Conference on Computer Science*, 1999.

[Lak92]   Arun Lakhotia. Improved interprocedural slicing algorithm, 1992.

[LC03]   Gary T. Leavens and Yoonsik Cheon. Y.: Design by contract with jml, 2003.

[Lea91]   Gary T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Softw.*, 8(4):72–80, 1991.

[LV97]   Filippo Lanubile and Giuseppe Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Trans. Softw. Eng.*, 23(4):246–259, 1997.

[LW87]   Jim Lyle and Mark Weiser. Automatic bug location by program slic-
            ing. In *Proceedings of the Second International Conference on Com-
            puters and Applications*, pages 877–883, 1987.

[Lyl84]   James Robert Lyle. *Evaluating variations on program slicing for de-
            bugging (data-flow, ada)*. PhD thesis, College Park, MD, USA, 1984.

[MC88]   B. P. Miller and Jong-Deok Choi. A mechanism for efficient debugging
            of parallel programs. *SIGPLAN Not.*, 23(7):135–144, 1988.

[MET02]   M. Morisio, M. Ezran, and C. Tully. Success and failure factors in
            software reuse. *IEEE Trans. Softw. Eng.*, 28(4):340–357, 2002.

[Mey92]   Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–
            51, 1992.

[MMM95]   Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and
            research directions. *IEEE Trans. Softw. Eng.*, 21(6):528–562, 1995.

[MMM97]   Rym Mili, Ali Mili, and Roland T. Mittermeir. Storing and retrieving
            software components: A refinement based system. *IEEE Trans. Softw.
            Eng.*, 23(7):445–460, 1997.

[MNM87]   Bertrand Meyer, Jean-Marc Nerson, and Masanobu Matsuo. Eiffel:
            object-oriented design for software engineering. In *Proc. of the 1st
            European Software Engineering Conference on ESEC '87*, pages 221–
            229, London, UK, 1987. Springer-Verlag.

[Moc07]   Audris Mockus. Large-scale code reuse in open source software.
            In *FLOSS '07: Proceedings of the First International Workshop on
            Emerging Trends in FLOSS Research and Development*, page 7, Wash-
            ington, DC, USA, 2007. IEEE Computer Society.

[MS93]   N. A. M. Maiden and A. G. Sutcliffe. People-oriented software reuse:
            the very thought. In *Advances in Software Reuse - Second Interna-
            tional Workshop on Software Reusability*, pages 176–185. IEEE Com-
            puter Society Press, 1993.

[MS03]   Tim Menzies and Justin S. Di Stefano. More success and failure fac-
            tors in software reuse. *IEEE Transactions on Software Engineering*,
            29(5):474–477, 2003.

[NE02]   Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of
            program specifications. In *ISSTA '02: Proceedings of the 2002 ACM
            SIGSOFT international symposium on Software testing and analysis*,
            pages 229–239, New York, NY, USA, 2002. ACM.

[OO84]   Karl J. Ottenstein and Linda M. Ottenstein. The program dependence
            graph in a software development environment. In *SDE 1: Proceedings*

of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, New York, NY, USA, 1984. ACM.

[OSH01]   Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Incremental slicing based on data-dependences types, 2001.

[OT93]   Linda M. Ott and Jeffrey J. Thuss. Slice based metrics for estimating cohesion. In *In Proceedings of the IEEE-CS International Metrics Symposium*, pages 71–81. IEEE Computer Society Press, 1993.

[RDKN03]   Marcus A. Rothenberger, Kevin J. Dooley, Uday R. Kulkarni, and Nader Nada. Strategies for software reuse: A principal component analysis of reuse practices. *IEEE Trans. Softw. Eng.*, 29(9):825–837, 2003.

[Rep91]   Thomas Reps. Algebraic properties of program integration. In *ESOP '90: Selected papers from the symposium on 3rd European symposium on programming*, pages 139–215, Amsterdam, The Netherlands, The Netherlands, 1991. Elsevier North-Holland, Inc.

[RW91]   Eugene J. Rollins and Jeannette M. Wing. Specifications as search keys for software libraries. In *IN PROCEEDINGS OF THE EIGHTH INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING*, pages 173–187. MIT Press, 1991.

[RY89]   Thomas W. Reps and Wuu Yang. The semantics of program slicing and program integration. In *TAPSOFT '89: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2*, pages 360–374, London, UK, 1989. Springer-Verlag.

[SF97]   Johann Schumann and Bernd Fischer. Nora/hammr: making deduction-based software component retrieval practical. In *ASE '97: Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*, page 246, Washington, DC, USA, 1997. IEEE Computer Society.

[Sha95]   Mary Shaw. Architectural issues in software reuse: it's not just the functionality, it's the packaging. *SIGSOFT Softw. Eng. Notes*, 20(SI):3–6, 1995.

[Sim88]   M. A. Simos. The domain-oriented software life cycle: towards an extended process model for reusability. pages 354–363, 1988.

[SS07]   Sajjan G. Shiva and Lubna Abou Shala. Software reuse: Research and practice. In *ITNG*, pages 603–609. IEEE Computer Society, 2007.

[SV03]   Karma Sherif and Ajay Vinze. Barriers to adoption of software reuse a qualitative study. *Inf. Manage.*, 41(2):159–175, 2003.

[SW94]   Marulli Sitariman and Bruce Weide. Component-based software using resolve. *SIGSOFT Softw. Eng. Notes*, 19(4):21–22, 1994.

[TBmJ06]   Yves Le Traon, Benoit Baudry, and Jean marc Jézéquel. Design by contract to improve software vigilance. *IEEE Trans. Softw. Eng*, 32:2006, 2006.

[Tip94]   Frank Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[Tra95]   Will Tracz. Dssa (domain-specific software architecture): pedagogical example. *SIGSOFT Softw. Eng. Notes*, 20(3):49–62, 1995.

[Tsi91]   Christine L. Tsien. Automated link creation in a hypertext-based software reuse library. 1991.

[Wei79]   Mark David Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, Ann Arbor, MI, USA, 1979.

[Wei81]   Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[WPF+10]   Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *ISSTA '10: Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72, New York, NY, USA, 2010. ACM.

[XQZ+05]   Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[Ye01]   Yunwen Ye. An active and adaptive reuse repository system. In *in Proceedings of 34th Hawaii International Conference on System Sciences (HICSS-34), IEEE Press, Maui, HI*. Press, 2001.

[ZG04]   Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 94–106, New York, NY, USA, 2004. ACM.

[ZGZ03]   Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 319–329, Washington, DC, USA, 2003. IEEE Computer Society.

[Zha98]   Jianjun Zhao. Applying slicing technique to software architectures. In *In Proc. of 4th IEEE International Conferencei on Engineering of Complex Computer Systems*, pages 87–98, 1998.

[ZW96]   Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6:333–369, 1996.

[ZXG05]   Yingzhou Zhang, Baowen Xu, and Jose Emilio Labra Gayo. A formal method for program slicing. In *ASWEC '05: Proceedings of the 2005 Australian conference on Software Engineering*, pages 140–148, Washington, DC, USA, 2005. IEEE Computer Society.