UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

GABRIEL MAIER FERNANDES VIDUEIRO
PEREIRA

Test-case-based Call Graph Construction in Dynamically Typed Programming Languages

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Marcelo Pimenta
Coadvisor: Profa. Dr. Ingrid Nunes

Porto Alegre
September 2015

CIP – CATALOGING-IN-PUBLICATION

Pereira, Gabriel Maier Fernandes Vidueiro

Test-case-based Call Graph Construction in Dynamically Typed Programming Languages / Gabriel Maier Fernandes Vidueiro Pereira. – Porto Alegre: PPGC da UFRGS, 2015.

46 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2015. Advisor: Marcelo Pimenta; Coadvisor: Ingrid Nunes.

1. Dynamic Languages. 2. Source code analysis. 3. Call Graphs. 4. Ruby. I. Pimenta, Marcelo. II. Nunes, Ingrid. III. Título.

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

ABSTRACT

Evolving enterprise software systems is one of the most challenging activities of the software development process. An important issue associated with this activity is to properly comprehend the source code and other software assets that must be evolved. To assist developers on these evolution tasks, Integrated Development Environments (IDEs) build tools that provides information about the source code and its dependencies. However, dynamically typed languages do not define types explicitly in the source code, which difficult source code analysis and therefore the construction of these tools. As an example, the call graph construction, used by IDE's to build source code navigation tools, is hampered by the absence of type definition. To address the problem of constructing call graphs for dynamic languages, we propose a technique based on steps to build a call graph based on test runtime information, called test-case-based call graph. The technique is divided in three steps; *Step #1* creates a conservative and static call graph that decides target nodes based on method names, and the first step also run tests profiling its execution; *Step #2* combines the test runtime information and the conservative call graph built in the first step to create the test-case-based call graph, it also creates a set of association rules to guide developers in the maintenance while creating new pieces of code; Finally, *Step #3* uses the test-case-based call graph and the association rules to assist developers in source code navigation tasks. Our evaluation on a large-size real-world software shows that the technique increases call graph precision removing several unnecessary conservative edges (%70), and assist developers filtering target nodes of method calls based on association rules extracted from the call graph.

Keywords: Dynamic Languages. Source code analysis. Call Graphs. Ruby.

RESUMO

Evolução de software é uma das atividades mais desafiadoras do processo de desenvolvimento de software. Uma importante questão associada à essa atividade é a correta compreensão do código fonte e outros artefatos que necessitam ser mantidos e evoluídos. Visando auxiliar desenvolvedores na manutenção de código, Integrated Development Environments (IDE's) proporcionam ferramentas que informam desenvolvedores sobre as dependências e as particularidades do código a ser modificado. No entanto, linguagens dinamicamente tipadas não definem tipos explicitamente no código fonte, o que dificulta a análise estática do código e consequentemente a contrução dessas ferramentas. Como exemplo, a construção de call graphs (grafos de chamadas), utilizados pelas IDE's para criar ferramentas de navegação de código, é prejudicada pela ausência da definição de tipos. Para abordar o problema da criação de call graphs para linguagens dinamicamente tipadas, propomos uma técnica dividida em passos para a construção de um call graph baseado em informações extraídas da execução de testes. A técnica é dividida em 3 passos, o *Passo #1* cria um call graph conservativo e estático que resolve chamadas de métodos baseado apenas em nomes dos métodos, ainda no primeiro passo, testes são executados e seu traço de execução é armazenado para posterior análise. O *Passo #2* combina a informação armazenada da execução dos testes e o call graph construído no primeiro passo, o *Passo #2* também é responsável pela criação de um conjunto de regras de associação que servirão para guiar desenvolvedores durante a criação de novas partes do código. Por último, o *Passo #3* utiliza o test-case-based call graph e as regras de associação criados no segundo passo para auxiliar desenvolvedores na navegação de código. Nossa avaliação em uma aplicação real de porte grande mostrou que a técnica melhora a precisão do call graph criado removendo arestas desnecessárias (70%), e mostrou-se apta a auxiliar desenvolvedores definindo pontos de navegação no código baseada na análise de regras de associação extraídas do test-case-based call graph.

Palavras-chave: Linguagens dinâmicas, Análise de código, Grafo de chamadas, Ruby.

LIST OF ABBREVIATIONS AND ACRONYMS

BPMN	Business Process Model Notation
CG	Call Graph
CSS	Cascading Style Sheets
CRM	Customer Relationship Manager
GQM	Goal Question Metric
HTML	HyperText Markup Language

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Evolving enterprise software systems is one of the most challenging activities of the software development process (MENS et al., 2005). An important issue associated with this activity is to properly comprehend the source code and other software assets that must be evolved. This comprehension process often requires to correctly identify the dependencies among different parts of the source code. Knowing dependencies is also essential to analyse the impact of changes (BOHNER, 1996). A key form of dependency is the invocation calls that occur within the source code, for example, in object-oriented software a method of class may invoke a method of another. This invocation dependency is often represented in a human readable graphical format which are referred to as *call graph (CG)* (GROVE; CHAMBERS, 2001) (RYDER, 1979). In this graph structure, methods are represented as nodes, while invocations are represented as edges. A CG makes the invocations between all methods in a software system explicit, and can be used to navigate within the source code, facilitating its understanding, as well as to analyse the change impact.

Building CGs may seem easy, because in some cases calls can be directly retrieved from the source code. However, there are issues that impose challenges for the CG construction. The first challenge is *inheritance and polymorphism*. When a method declared in an interface is invoked, there may be different implementations that can actually be executed as a consequence of this call. Therefore, it is often impossible to know with solely static information which of the implementations will be actually be executed given the call. The second challenge is *dynamically typed programming languages*, such as script languages like the popular Ruby, Python and JavaScript languages. In this case, program variables may hold objects of any type, therefore it is also not obvious which implementation will be executed when methods are invoked. In order to create CGs for script languages, for example, IDEs consider only method names, ignoring (actual possible) types of objects that make a call. Consequently, if more than one class has the same method name, it is not possible to decide the target node of the CG. The conservative approach to build a CG is thus to add an edge to each possible target, which decreases the precision of the CG given that it will contain unnecessary edges.

Different approaches have been proposed to support the construction of CGs for dynamically typed languages (FELDTHAUS et al., 2013) (FURR et al., 2009) (ANDERSON; GIANNINI; DROSSOPOULOU, 2005), but they focus on assist the construction of *static* CGs, i.e. they use static information retrieved from the source code to build the CG, by searching for method definitions and method calls and performing *type inference*, for example. However, it is

not always possible to build a precise CG. An alternative is to build *dynamic* CGs, where source code needs to be executed to identify edges from the CG. Although a dynamic call that occurs when executing the software guarantees that an edge must be added to the CG, dynamic CGs are often not exploited due to the need for exercising all different paths in the source code to achieve high levels of recall. Consequently, runtime analysis of source code is a *path-sensitive* technique, which means that the output of the analysis is proportional to the size of the executed code. Thus, covering all possible executions of the source code is practically impossible or at least too expensive to be done. Moreover, capturing calls during runtime requires code instrumentation, which degrades software performance since it is necessary to wrap all code execution to perform the analysis. Therefore, dynamic CGs should be used conscientiously trying to balance between performance and analysis coverage. Even though building dynamic CGs has these issues, the trend of adopting agile methods for software development can be helpful to overcome them. One of the key practices of agile methods is using a test-driven development approach (BECK, 2003), in which automated test cases are written and available to be executed. Therefore, such test cases can be used as input to built dynamic CGs.

We thus propose an approach to support the maintenance of software developed using dynamically typed programming languages by facilitating navigation in the source code using dynamically-enriched CGs, exploiting available test cases. Our approach consists of a set of steps to be integrated as part of a test-driven development process. The key idea is to build a static CG based on the source code and then, by executing test cases offline, extract data to enrich this CG with dynamically-extracted information. These data are used to annotate the edges of the CG with the likelihood that a call occurs at runtime, consequently increasing the precision of the CG. These enriched CGs are useful to facilitate code navigation in IDEs, as well as to analyse and visualise code dependencies de facto. In order to validate our approach, we performed an empirical study using software written in Ruby, developed within the context of a large-size software company. We collected information from developers while they were maintaining the software, and used it to evaluate the precision of our CG. Results indicate that our approach is effective and has the potencial to support the aforementioned activities associated with the development of software written with dynamically typed programming languages.

The dissertation structure is as follows. We start discussing related work in Section 2. We then provide background on the construction of call graphs for both statically and dynamically typed programming languages in Section 3. Our approach is then described in Section 4 and its evaluation is presented in Section 5, before we conclude in Section 6.

2 RELATED WORK

Javascript is one of the most popular dynamic typed programming language. Despite its popularity, a sound static analysis it is still a challenge, as for other dynamic typed languages. Feldthaus et al.(FELDTHAUS et al., 2013) presented a field-based flow analysis for Javascript to compute approximate call graphs that scales well to real world applications for usage in IDEs. Similarly to our approach, for method call `o.m()`, their technique cannot reason about the type of `o` variable and, therefore, consider all implementations of `m()` as possible target methods. They propose and evaluate two approaches, one pessimist that only tracks interprocedural flow and a optimist approach that tracks interprocedural and intraprocedural flows. Despite the good results, mainly of the pessimistic approach, they propose the construction of an approximative CG based on a static only analysis of a dynamic language, which hamper good *recall* results. In turn, we proposed a hybrid approach to improve the CG dynamically based on test execution on top of a static and conservative previous analysis. Therefore, both approaches can be used together to probably achieve better results while combining static and dynamic approaches.

Dynamic test information was used recently by Toma et al. (TOMA et al., 2015) to build a dependency graph for Javascript, HTML and CSS. Based in a dynamic call graph construction for Javascript, a dependency graph is created interconnecting function calls with HTML nodes and CSS definitions. Their approach is a *dynamic only* analysis based on end-to-end test execution to build a call graph based on actual method calls that happens during tests. The *dynamic only* analysis limits their results to the set of paths executed in the test environment. In fact, to improve the coverage of the test environment, and consequently the call graph precision, a set of additional tests were created by a experienced developer to better evaluate their technique. Our proposed technique also depends of a good test coverage, however we evaluate our technique against the existent test environment to better approximate of a real world scenario. Moreover, our static call graph creation step ensure that edges are added in the CG even if the test environment does not cover a particular method call, and so, our technique is not so dependent of the dynamic execution.

To help building calls graphs and other tools that assist developers, different approaches aim at static type inference analysis for dynamic typed languages (ANDERSON; GIANNINI; DROSSOPOULOU, 2005) (CANNON, 2005). These techniques analyzes source code statically or dynamically aiming at to infer about variable types. At this way, mature source code analysis tools that exists for typed languages like Java and C# could be adapted for dynamic languages, including call graph creation. Aiming to ease type inference systems, source code

annotations are used to explicitly define types to variables and functions (FURR et al., 2009). Furr et al. also used runtime information to extend their previous work; they used the profiled analysis of runtime to guide a transformation phase that replaces dynamic constructs with static constructs specialized to the values seen at run time to make the code statically typable. (FURR; AN; FOSTER, 2009). Finally they also concludes that the usage of runtime information can enhance static analysis of dynamically typed languages. Differently from a type inference system, our approach takes advantage of the knowledge coded in tests to build a call graph based on actual types extracted from runtime information.

Despite the fact that static typed programming languages eases the call graph construction due to type checking systems, these languages also needs some approximation of the target methods due to language constructions like virtual method calls. Heintze et al.(HEINTZE; TARDIEU, 2001) used Points-to analysis to approximate these target methods in C programming language; later Lhoták et al. (LHOTÁK; HENDREN, 2003) extended this approach to handle object orientation characteristics. Dynamic languages are also subject of Points-to analysis (JANG; CHOE, 2009). Differently from Points-to analysis approach, we propose a CG construction without source code flow analysis since the time consumption is a constraint while building tools for IDEs.

3 BACKGROUND ON CALL GRAPHS

In this section, we provide background on call graphs, discussing challenges association with their construction. A call graph (CG) is a directed graph that represents possible execution flows of a program. Formally, it is a graph $G = \langle V, E \rangle$, where V is a set of nodes that represent entities able to call other entities, and E is a set E of edges that represent calls between entities. In object-oriented languages, which is the focus of this work, entities are *methods* of classes, while edges represent method calls between methods.

Given this CG definition, we next detail, in Sections 3.1 and 3.2, how CGs are constructed for statically typed programming languages (hereafter, referred to as typed languages) and for dynamically typed programming languages (hereafter, referred to as dynamic languages), respectively.

3.1 CG for Statically Typed Languages

Building CGs for typed languages, such as Java and C#, may seem easy since method calls and their target can be retrieved directly from the source code. The statically-typed nature of these languages forces developers to specify types for each variable, and consequently enabling the inference of the target of each method call from the source code, which eases static analysis when creating CGs. Although this is true for some method calls, there are issues involved with the creation static CGs, that is, those built using static analysis.

Static CGs for typed languages are built as follows. For each method definition (i.e. entities) present in the source code, a node is added to the CG. As said above, edges represent method calls, so all method calls are analyzed and, for each method invocation, an edge is inserted in the graph, starting from the caller method and targeting the callee method, as shown in Figure 3.1. In order to correctly identify target methods of a method call, it is necessary to know the type of the variable that is used to make the invocation, which is not difficult in typed languages due to need for defining types for all variables in the source code. However, object-oriented language structures such as *polymorphism* and *inheritance* imposes a difficulty to target identification, since they allow different method implementations to be a target of the same call. The choice for the actually used implementation is made at runtime. However, in some cases, a developer may know that, from the possible implementations, only a subset is actually the target of a particular call. Therefore, building CGs for typed languages implies handling two types of calls: (i) *simple calls*, whose target can be directly retrieved from source code; and (ii) *polymorphic calls*, which require a more powerful analysis to infer possible target methods of a method call.

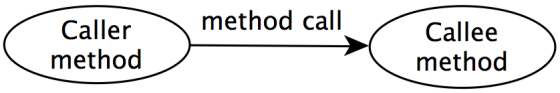

Figure 3.1 – Representation of a method call in a CG.

Both these types of calls, *simple* and *polymorphic*, are shown in Figure 3.2, where a simplified Java code is used as an example to illustrate the relationship between a source code and its CG. A *simple call*, e.g. `gsPanel.display()` (line 5), can be easily retrieved from source code since there is no ambiguity about which method should handle this call. Since the variable `gsPanel` has the known type `GeoShapePanel`, which is a concrete class declared in lines 8–13, there is no other method that can handle this call than the `GeoShapePanel.display()` method (line 10). The resulting CG for this code is shown in the left hand side of Figure 3.2, in which the edge **a** represents the `gsPanel.display()` call. Since this call takes place in the `main` method, the edge goes from the *main* node to the *GeoShapePanel.display()* node, which is the only possible implementation of this simple call.

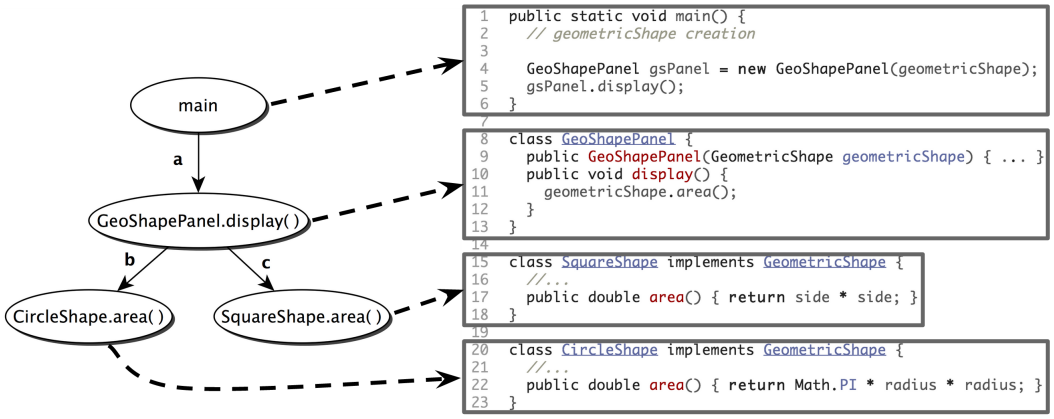

Figure 3.2 – Java example

Simple calls are easy to analyze; polymorphic calls, in turn, adds uncertainty in the resulting CG since it is difficult (or even impossible) to infer target nodes only with source code static analysis. The `geometricShape.area()` call (line 11) is an example of a *polymorphic* call since the `area()` method is defined in the `GeometricShape` interface (omitted to simplify the diagram) and implemented by two different classes, `SquareShape` and `CircleShape` (shown in lines 15–18 and 20–23, respectively). The actual target node of the call depends on the `geometricShape` type that in the example must be instantiated in the main method (line 4). The `geometricShape` variable creation is omitted in the example since there are

two possibilities for the variable type instantiation, which impacts in the analysis. Firstly, if the `geometricShape` variable is instantiated with one of the `GeometricShape` implementations, such as `GeometricShape gs = new SquareShape()`, it is possible to infer that the edge **c** is the correct edge for this scenario since the variable would always be of the type `SquareShape`. However, the concrete `geometricShape` variable type could be only known at runtime if the instantiation is not hard coded, as exemplified above, and then it would be impossible to statically infer which one of the two edges (**b** or **c**) is the correct edge of the call. An example of this scenario is the use of a user input to choose how to instantiate the variable, shown on Figure 3.3, which makes it impossible to decide which one of the possible edges is correct without running the code — and in this case the adequate CG includes both **b** and **c**. This difficulty that is associated with static analysis can be overcome using a dynamic analysis approach, which monitors the execution to remove edges that do not correspond to calls made at runtime.

```
1 user_input = IO.read_input();
2
3 if (user_input == 'Y') {
4   GeometricShape gs = new SquareShape();
5 } else {
6   GeometricShape gs = new CircleShape();
7 }
```

Figure 3.3 – User input defines variable type

Despite the fact that *polymorphic calls* are a challenge to construct CGs for typed languages, the interface type definition limits the amount of uncertainty of a specific method call. For instance, Figure 3.4 shows a diagram that presents two interfaces with 3 different method implementations each. In this scenario, an invocation to a method called `read` that implements the interface `File` will have 3 different possible target nodes, shown in the left hand side of the image. In turn, an invocation to method `read` implemented by the interface `Document` will have the other 3 possible target nodes, shown in the right hand side of the image. Although polymorphism difficult static analysis of typed language, the problem is diminish by the interface type definition. In the other hand, dynamic languages do not have explicit interface definition, therefore the same method call to a method `read` would have all the 6 implementations as possible target nodes.

Even though tools that support source code analysis for typed languages have significantly improved over the years, the construction of static CGs is still a challenge due to above discussed points. In dynamic languages, these problems remain unsolved. Moreover, due to the

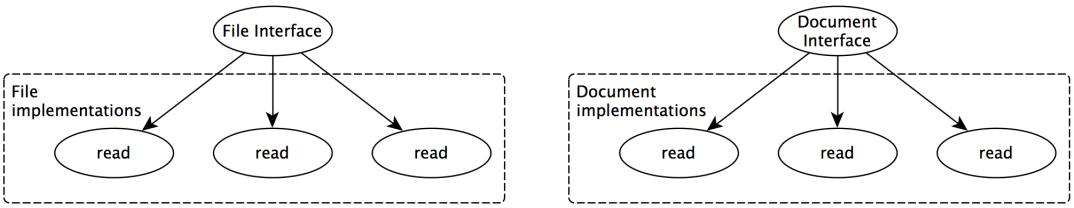

Figure 3.4 – Multiple interfaces with same method name

flexibility of creating variables without binding variable types, this problem is even worse, and its particularities are discussed in the next section.

3.2 CG for Dynamically Typed Languages

Dynamic typed languages are becoming more popular nowadays. Their dynamic nature gives more freedom to developers to create and use objects without the need for declaring types of each variable and method parameters and return (TRATT, 2009). Moreover, practitioners argue that dynamic languages increase their productivity since it is easier to code tests and it is less verbose to implement object oriented concepts. Apart from developers personal beliefs, the number of dynamic typed languages adopters is increasing (TIOBE. . . ,) and sound source code analysis tools to build CGs are still missing.

To build static CGs it is necessary to know variable types to correctly specify target nodes of method calls; however, dynamic typed languages do not make type definitions in source code explicit and, consequently, type inference systems or conservative approaches must be chosen. Type inference systems, which are a more precise alternative, inspect source code by searching for possible flows of execution and variable assignments that allow to infer an object type. Approaches based on control-flow and data-flow analysis can be combined with type inference systems to improve their results. Nevertheless, the absence of a type definition in dynamic languages increases the complexity of inferring types since variables and methods are not bounded to an explicit type, forcing algorithms to process all possible types. In order to reduce the complexity of this problem, some approaches use source code annotations to inform method and variable signatures to perform type inference with more information than solely the source code (AN et al., 2011). In spite of the fact that code annotations improve the precision of static analysis, it does not solve the problem of maintenance and evolution of legacy code, since developers must know details of the code that she wants to understand. Moreover, there is not guarantee that annotations are updated as the code evolves. It is worth to mention that

even with a sound type inference system, it is not possible to accurately create a complete static CG for dynamic languages since inherits the issues associated with typed languages, discussed above.

As an alternative to source code annotations and time consuming analysis made by type inference systems, conservative approaches to build CGs can be used to assist developers in maintenance tasks performed in Integrated Development Environments (IDEs). Conservative approaches address the problem of choosing target methods of a call by adding edges to all possible methods in the source code. The set of possible target methods is selected only by evaluating method names, in contrast to conservative approaches for typed languages in which both the method name and variable types are considered in the analysis. To illustrate, a conservative approach for dynamic languages considers all implementations of a method `m()` as target methods of a method call `o.m()`, without considering the type of the `o` variable. Despite the good performance of a conservative approach, the algorithm simplicity creates too many conservative edges in CGs, thus decreasing their precision.

Conservative edges inserted in CGs due to method name collisions can be split into two groups: collisions by *design decision* and collisions by *coincidence*. Collisions by design decision are the same aforementioned undecidable target nodes of typed languages, where it is not possible to decide object types statically due to *inheritance* or *polymorphism* usages. Typically, this is the case when methods with the same name represent a common interface. Conversely, collision by coincidence occurs by chance, and methods with the same name have only syntactic similarity, and do not have any semantic relationship. In this case, when navigating in the code, developers get confused with many irrelevant alternatives for a specific method call.

To exemplify occurrences of both groups of collisions above mentioned, a medium-size open source application built in a dynamic language is presented. FatFreeCRM[1] is a customer relationship manager tool with 4K lines of Ruby code, composed of 65 classes, 387 methods and 1,839 method invocations. FatFreeCRM is built in a popular dynamic language and despite of its medium size it is possible to observe in its code both types of introduced collisions.

As in typed languages, collisions by design decision occur when there is an explicit intent of creating methods that implements a similar contract (interfaces on typed languages), and so same method name is chosen. As opposed to typed languages in which explicit interfaces can be defined, most dynamic languages do not provide the concept of interface, and so interfaces are implicitly specified. As an example of implicit interfaces, Figure 3.5 shows a CG example of collisions by design decision where 5 different classes implement an implicit interface

[1] <http://www.fatfreecrm.com/>

that specifies the behavior of attaching something to an object. The method `attach!` is implemented by these 5 different classes and all implementations share the same responsibility, but have individual implementations. Due to this reason, it is not possible to statically identify which one of the possible target methods is the desired method of the `Controller.attach` call. In spite of the impossibility to decide target nodes for design decision collisions, the edges added to the CG by this type of collisions help developers to understand the possible variable types when analyzing a method call. In addition, this give an indication that adequate tests must cover all possible implementations. In fact, even using an IDE with static code analysis (Rubymine[2]) is not possible to infer about the target node of this particular call as it is shown in Figure 3.6. However, it is worth mention that a runtime analysis can assist developers pointing out the target nodes that are actually called from `Controller.save` method, since some of the `attach!` implementations may not make sense in the `Controller.save` context.

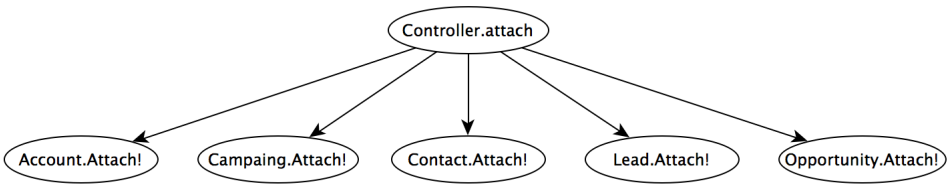

Figure 3.5 – Example of Collision by Decision.

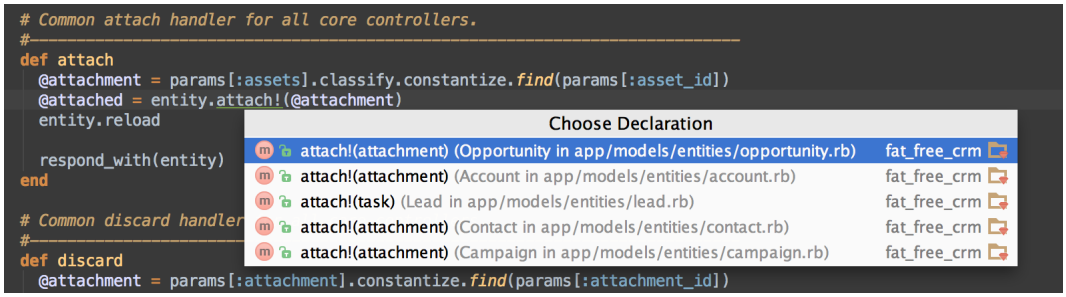

Figure 3.6 – Suggestion list of attach! method call

Collisions by coincidence, in turn, negatively impact in maintenance tasks because it causes the addition of incorrect information to the CG, leading to a misbehavior on the *jump-to-declaration* functionality. As an example of this problem, Figure 3.7 shows a graph where two methods are named `current_user`. They both are considered as target methods of a call to this method, but they have different responsibilities in the application. The *ApplicationController.current_user* node, in the left hand side, is responsible for managing the user session,

[2]https://www.jetbrains.com/ruby/

while the node in the right hand side, *EntityObserver.current_user*, is responsible for fixing an issue with an external library. Typically, IDEs show these two methods as possible target nodes of the call performed in the `links_to_export`, as shown in Figure 3.8, despite the fact that each method has a particular scenario where it makes sense to be called. Even though almost all method calls to `current_user` made in the code aims to `ApplicationController` class, both options are suggested equally to developers; to overcome this problem, a runtime analysis can be used to enhance the suggestion list order.

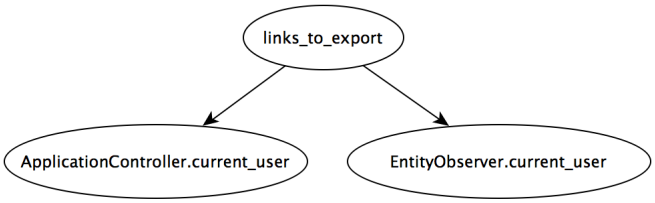

Figure 3.7 – Example of Collision by Coincidence: Simple Case.

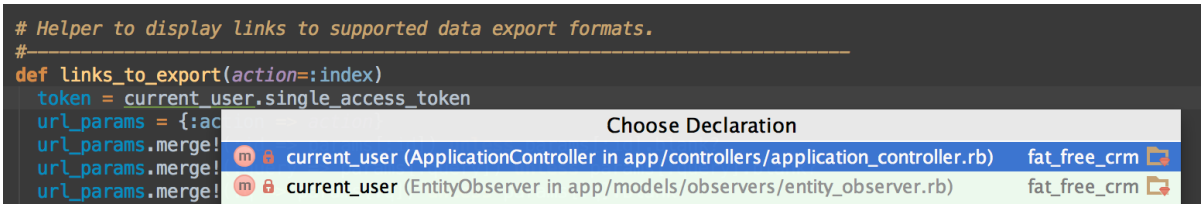

Figure 3.8 – Suggestion list for current_user method call.

A worse scenario where 16 different implementations of the method `create` are considered as target nodes of a method call is shown in Figure 3.9. Since *create* is a pretty common method name in libraries and applications, IDEs that use some kind of static code analysis make the suggestion list even worse than just conservative approaches, since they show as possible target nodes implementations from several external libraries, as Figure 3.10 shows. These scenarios clearly show the low precision of conservative approaches and the need for better tools to select appropriate target nodes in statically generated CGs for dynamic typed languages.

In the next section we present our proposed technique, which leverages information gathered during test execution. This improves the generation of static CGs, helping overcome the aforementioned problems that emerge due to collisions of method name by coincidence using conservative solutions.

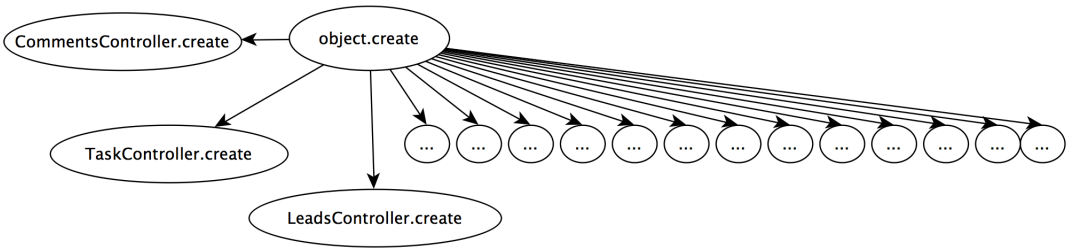

Figure 3.9 – Example of Collision by Coincidence: Complex Case.

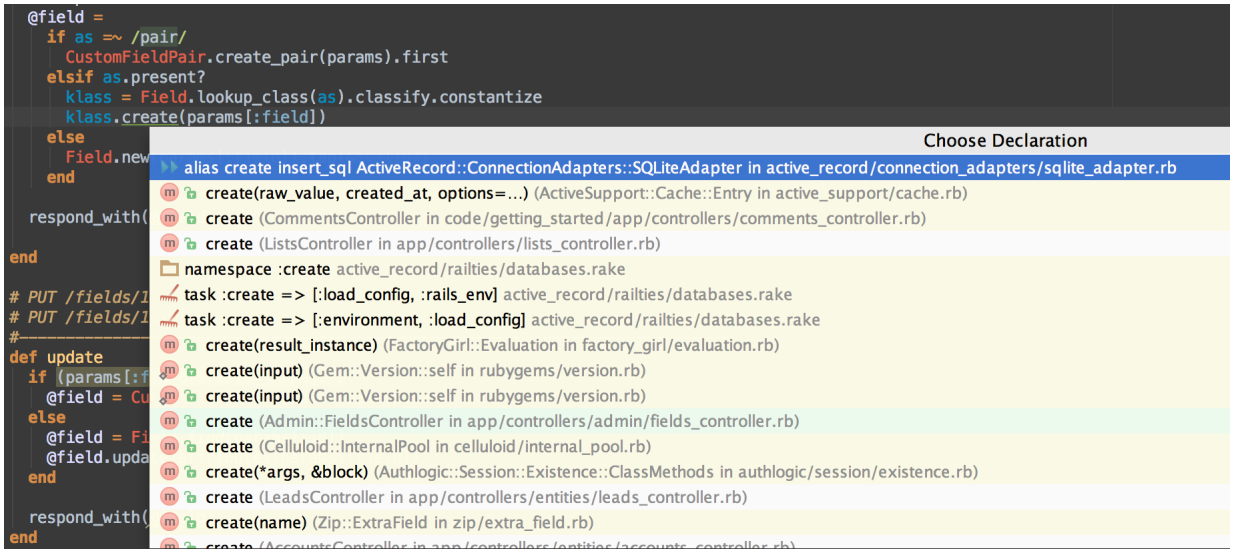

Figure 3.10 – IDE's suggestion list for create method.

4 A TECHNIQUE TO CONSTRUCT CALL GRAPHS BASED ON TEST CASES

To overcome the aforementioned problems of call graph construction and source code navigation for dynamic languages, the proposed technique relies in a set of steps that benefits from data extracted of dynamic test execution on software pipelines. All computation happens on pipeline machines, avoiding over processing in developing environments, and the outputs (Test-Case-Based CG and Association Rules) are shared as artifacts for IDEs to consume and guide developers on maintenance and software evolution tasks.

In a nutshell, we first execute automated tests profiling its execution and separately build a CG using a conservative approach (Step #1). Second, based on executed tests, we annotated the generated CG and derive rules that associate a context to an implementation used (Step #2). These two steps are performed offline. And third, we use this information to navigate within both existing code and code being written (Step #3). All these steps and their tasks are shown in the diagram presented in Figure 4.1, using the BPM notation (WHITE, 2004). In the next subsections, each step of the technique is explained in details.

4.1 Step #1: Profiling tests and conservative CG creation

Dynamic typed languages are hard to analyze statically since variables, function parameters and return types can change at runtime. Runtime analysis, in turn, has the ability to capture actual types and method calls at runtime enabling a better analysis of the source code. However, runtime analysis is a path sensitive task and the quality of outcome is highly dependent on the amount of executed software paths. While covering all possible execution paths of a system could be considered impossible or at least too expensive to be done, software paths should be chosen conscientiously to avoid misconceptions about the source code behavior. To overcome the problem of choosing relevant software paths to be analyzed, the proposed technique use the knowledge of experts in the system to guide the path selection. This expertise of relevant paths is extracted from automated test scenarios that are considered important for the system by developers that know the domain; that is, if a software path is important enough to have a test exercising it, it should be important enough to guide developers in the source code comprehension and software maintenance tasks.

Aiming at the construction of a test-case-based call graph, the first step of the technique is composed of two parallel tasks called *Run Tests* and *Build Call Graph*. The *Run Tests* task is responsible for storing runtime information of automated test executions on software pipelines through source code profiling. At this way, each test case is an entry point of analysis where the

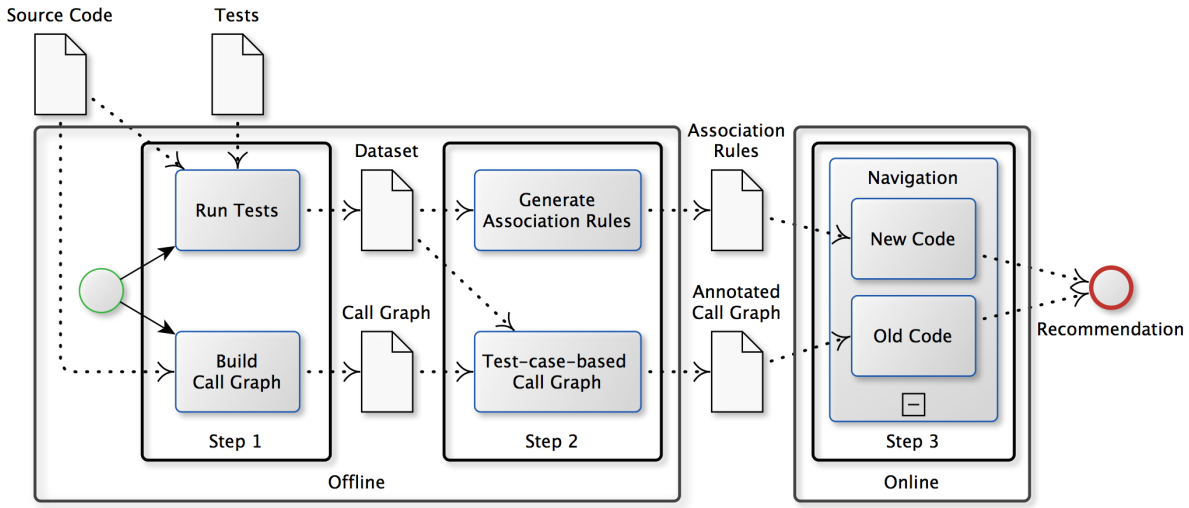

Figure 4.1 – Technique Steps.

profiling tool stores information about the application runtime. The *Run Tests* is the key task of the whole process as it creates a reliable set of data about actual method calls that are used as input for the subsequent steps of the technique; also the accuracy of next steps are highly dependent on the quality of the *Run Tests* task output.

To collect data related to test runtime it is necessary to profile the piece of code under test. Profiling source code is a common approach to investigate how an application behaves at runtime, being performance and trace analysis common examples of its usage. Source code profiling consists of wrapping the code under analysis by a routine that inspect the execution stack trace of a program to get the information about method calls made on runtime. The wrapper routine has an entry point, where it starts inspecting the program stack trace, and an exit point in which it stops the inspection and stores the extracted data. At this way, a piece of code under analysis should be placed between the entry and exit points and these points could be added directly in the application source code or in an external artifact that runs the application. In the proposed technique the profiling tool is added surrounding automated tests of an application, and so it profiles all software paths of execution that are covered by automated tests. Unlike popular profile analysis that usually harms application performance, this approach does not impact the performance of production environments since it only profile the code in test environments. Its execution can be scheduled to convenient times, which do not interfere the development.

Despite the simplicity of a stack trace analysis, there are some challenges to profile an application having tests as entry points of analysis since different granularities of tests require

different profiling strategies. On the one hand, fine-grained tests, like unit and integration tests, assert a particular piece of code isolated from the rest of the application. For this reason, fine-grained tests do not require the entire application to be executed, and so the test engine only loads the necessary parts of the application to run the test. This load strategy creates a shared stack trace between the test engine and the application, which allows to profile the application through the test environment since the test engine have access to the application stack trace. At this way, the entry point for analysis of fine-grained tests is the test library where each test scenario is wrapped by a profiling routine. Regardless of the easiness of profiling fine-grained tests, this kind of tests does not make many method calls during runtime due to the use of mimic strategies to create an isolation between the piece of code under test from the rest of the application. For instance, when testing a class A that uses methods from a class B, it is considered a good practice to mimic the behavior of B by creating a *double* (MESZAROS, 2007) to isolate the test of A from B's implementation details. The use of *doubles* improves the maintainability of the tests since it avoids ripple effect of changes when changing a class contract. However, this strategy masks the real relationship between classes since it believes on mimic behaviors instead of actual method calls. Although it is possible to capture *double* definitions on test scenarios and use this information to assert method calls, this information could be wrong or outdated since tests do not fail when contracts of a mocked class changes. For this reason, mimic strategies are not considered in the proposed solution, and so fine-grained tests tend to not make many method calls during tests when compared with end-to-end tests.

On the other hand, end-to-end tests asserts the application behavior in a coarse-grained level, and for this reason it is necessary to have the application running with the full environment loaded to execute tests. The requirement of having the application running implies that the test environment should run in a different system process to emulate the user interaction with the system. However, the application running in a different process does not create a shared stack trace with the test environment that would allow to profile it through the test engine as it happens with the fine-grained tests aforementioned, and so an alternative strategy should be applied. As a first alternative, it is possible to profile the entire application running in a server with a profiling flag enabled, which generates output data with information about every method call made in the server while it runs. As a second option, it is possible to take advantage of a particular software architecture to wrap all the executions in the server. For instance, several frameworks use the popular Model View Controller (MVC) (GAMMA et al., 1994) architecture where the *Controller* layer is responsible for handling every application request and delegating to the proper *Model*. At this way, each request to the server could be profiled by wrapping the

Controller layer of the application gaining access to the stack trace, and so, enabling the access of method calls made at runtime. Regardless the way chose to profile end-to-end tests, this type of tests tend to capture more method calls at runtime when compared to fine-grained tests, since coarse-grained tests do not use mimic strategies to isolate tests. However, end-to-end tests are expensive in time consumption, and so applications usually have less end-to-end tests than fine-grained tests, which conforms with the advise to create a test pyramid with fine-grained tests in the base and end-to-end at the top (BECK, 2003).

All the information profiled in the *Run Tests* task should be stored in a structure that helps developers on software comprehension tasks; call graphs (CG) are commonly used for this purpose since they are human readable and make software dependencies explicit and easy to understand. However, the construction of a CG based solely in the test execution could not be sound enough since test scenarios potentially do not cover all possible paths of a system, and so it is necessary to include method calls not covered by tests in the resultant CG. However, it is not straightforward to define the target node of a call `o.m()` when multiple classes implement a method called `m()` since there is no type definition of `o` in the code. To overcome this problem, a static and conservative call graph is created based on a field-based technique (FELDTHAUS et al., 2013) that only considers method names to define target nodes in the CG, which results in one edge targeting each existent `m()` implementation in the above example. The algorithm used to build the CG is shown in Algorithm 1. Since the field-based technique implies in a conservative analysis, problems related to *inheritance*, *polymorphism*, methods with the same signature and dynamic typing particularities (which were all discussed in the previous section) are not handled at this point of the technique, which results in a CG with multiple conservative edges for each undecidable target method. However, the conservative CG is used as a base for the next steps of the technique that would enhance the CG with test runtime information. Although conservative approaches create too many unnecessary edges in the CG, it is better to have extra edges in the CG than to forget an important edge and mislead developers. Moreover, the next steps of the technique handle this problem annotating the resultant CG with test execution data, thus improving its accuracy about actual method calls.

4.2 Step #2: Annotate CG & Generate Association Rules

Call graphs are used by IDEs to implement functionalities that assist developers in maintenance tasks like *Go to declaration*, *Source Code Refactoring*, and many others. However, IDEs for dynamic languages usually use conservative CGs to build these functionalities which mis-

```
1  callGraph ← ∅;
2  classes ← FindClasses(source_code);
   // First Round - Search for nodes
3  foreach class of classes do
4  |  methods ← FindMethods(class);
5  |  callGraph ← AddNodes(methods)
6  end

   // Second Round - Search for edges
7  foreach method present on callGraph do
8  |  invokedMethods ← FindInvocations(method);
9  |  foreach invokedMethod of invokedMethods do
10 |  |  if callGraph include invokedMethod then
11 |  |  |  callGraph ← AddEdge(method, invokedMethod)
12 |  |  end
13 |  end
14 end
```

Algorithm 1: Static call graph algorithm

lead developers showing several wrong edges in the graph as possible target nodes of a method call. To overcome this problem, different approaches could be chosen to enhance the CG soundness, like data-flow, control-flow and runtime analysis. Each one of the options has its pros and cons, but it is notable the contrast of the time consumption of the runtime analysis versus its soundness, since it bases the analysis in real code execution. The proposed test-case-based CG takes advantage of a test runtime analysis to annotate a CG with actual method calls aiming to improve its soundness, also reducing the issues with time consumption due to the use of test environments to perform the runtime analysis.

In the second step, the *Annotate Call Graph* task combines the outcomes of the first step to generate an enhanced CG with information about test execution to create what we refer to as test-case-based CG. The *dataset* output generated by the *Run Test* task is processed by searching for method calls that exist in the static CG generated by the *Build Call Graph* task. In other words, for each method call found in the *dataset* a counter is incremented in the respective CG edge. At this way, the conservative CG is annotated with actual data about method calls, where edges in the CG are labeled with a counter information with the number of occurrences of the respective method call during test runtime. The algorithm used to label edges in the graph is shown in Algorithm 2. As a result, the *Annotate Call Graph* task generates an enhanced CG with information about test runtime execution, and the different ways that this information assist developers while maintaining or evolving code is explained in details in Section 4.3. It is worth mentioning that this approach does not consider method calls that target methods created at

runtime since their method definition does not exist in the code, and so it does not help in the source code navigation.

```
1 edgesInTheGraph ← callGraphInvocations();
2 foreach profile of profilerFiles do
3 |   methodCalls ← methodCallsOf(profile);
4 |   foreach methodCall of methodCalls do
5 |   |   if edgesInTheGraph include methodCall then
6 |   |   |   callGraph ← incrementEdgeCounter()
7 |   |   end
8 |   end
9 end
```

Algorithm 2: Dynamic information on CG

Despite the fact that a Test-case-based Call Graph helps developers in maintenance tasks, the annotated CG is only available after a complete test environment execution under profiling analysis, which can take hours to run. At this way, due to the excessive processing time, it is not feasible to keep running tasks to create an annotated CG in local machines. However, popular practices in software engineering, like *Continuous Integration* and *Continuous Delivery* (FOWLER; FOEMMEL, 2006), encourage to use server machines to build, test and deploy applications continuously; and so it is possible to take advantage of these environments to profile tests and build the CG avoiding over processing in development workstations. These server machines work in a pipeline way where for each code pushed to the repository a *pipeline build* is automatically triggered with different jobs that have different responsibilities, like build, test and deploy the application. Thus, the test profiling task is handled by a pipeline job that keep an annotated CG constructed available as an artifact for each source code committed to the repository. Then, the test-Case-Based CG is shared as an artifact of the pipeline job allowing easy access to the annotated CG anytime needed without requiring processing in local machines.

Even though the test-case-based CG can be easily accessed, artifacts created in software pipelines become outdated as soon as a new piece of code is modified in a development environment; for instance the Test-case-based CG created in the last pipeline run has information about classes, methods and method calls that existed at that time. However, software evolves constantly in local machines while it is under development which creates the need for considering these source code modifications when assisting developers in maintenance tasks. One of the key problems happens when developers create a new method call that points to a polymorphic method, since there are multiple possible targets and there is no edge in the annotated CG that represents this new call. The existent annotated edges can consequently mislead de-

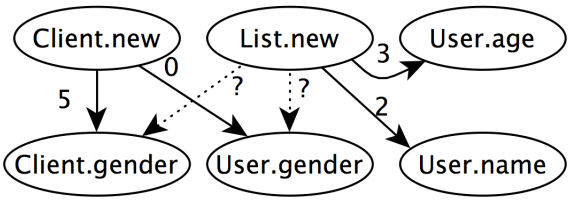

Figure 4.2 – New method call.

velopers showing wrong suggestions for target nodes. For instance, Figure 4.2 shows part of a test-case-based CG where the dotted line represents a new method call (edge) created after the last CG construction. Even tough the annotated CG tends do suggest `Client.gender` as a candidate target method, since it has 5 executions against zero executions of `User.gender` in test environments, the CG have an information that the caller method `List.new` usually calls methods from `User` class, and so, there is a probability that the `User.gender` method should be a better target method candidate. Aiming at a better approximation of possible target nodes, existing information stored in the CG can be used to help understand general trends in existent method calls. Thus, association rules (AGRAWAL; IMIELIŃSKI; SWAMI, 1993) are used to analyze the test-Case-Based CG and generate rules that exemplifiy the dependency of similar classes and methods with multiple implementations.

Association rules can be extracted from the test-case-based CG to make decisions regarding target methods of new code created after the last pipeline run without the need for running the full test environment. Since the annotated CG has actual information about method calls it is possible to derive rules about how methods interact with each other and then use these rules to assist developers when creating new source code. The *Generate Association Rules* task requires a dataset with relevant data to be analyzed and to derive rules, thus the dataset is built based on actual method call information stored in the test-case-based CG. Since method calls with multiple target nodes are problematic when dealing with CGs on dynamic languages, datasets are created focusing on these methods. To generate the rules, the Apriori algorithm (AGRAWAL et al., 1996) is used and the dataset is built as follows.

1. A dataset is created for each application class that implements at least one polymorphic method. The set of attributes in the dataset is composed of all application classes that invoke at least one polymorphic method.
2. For each dataset, *class association rules* are mined instead of general association rules. It means that the algorithm mine the dataset searching for item sets of attributes that derive

to the desired *target class*. Each dataset has a *target class* that is the application class.

3. While executing a test, each method call that targets a *target class* is added as a transaction in the dataset. For each transaction, the attributes are marked as TRUE or FALSE to inform which other classes are used by a class that invokes the *dataset target class*, i.e. if the class `A` has a method call to the *target class*, the attributes informs which other classes are called by the class `A`.

An example of dataset is shown in the Table 4.1, where the first line displays in columns the attributes: classes `A`, `B`, `C` and `D`; and lastly a column with the *dataset target class*. In the example, the *target class* is the application class `D`, and this line was added in the dataset as a transaction since a method call to `D.method()` occurred in the test environment. The existence of this line means that the class `D` has a polymorphic method, but `method()` does not need necessarily to be the polymorphic method. Also, the first flag FALSE in the column of `A` attribute informs that the caller of method `D.method()` does not calls any method of `A`. In turn, the TRUE flag in the column of `B` attributes informs that the caller of method `D.method()` also invokes in the test environment at least one method of the class `B`. The same logic applies to all attribute columns. This example dataset is composed of only one transaction represented by one line, but data mining algorithms usually require datasets with several transactions (lines) to be able to derive rules.

Table 4.1 – Dataset example.

A	**B**	**C**	**D**	**Target Class**
FALSE	TRUE	FALSE	TRUE	D.method()

The *Generate Association Rules* task creates a set of association rules that is available as an artifact to be consumed by IDEs. This is the last offline task of the proposed technique and the next step of the technique is composed of online tasks that are responsible for guiding developers in maintenance tasks based on the artifacts created in the offline tasks.

4.3 Step #3: Source code navigation: Test-case-based Call Graph

Maintaining and evolving source code is facilitated when it is possible to navigate in the code understanding software dependencies; however, this source code navigation is not an easy task in software written in dynamic languages given that it is not always possible to know all types statically. To assist developers in this important task, two approaches are presented. First,

the use of a test-case-based CG when navigating through existent source code, and secondly the use of Association Rules to assist developers while creating new code.

4.3.1 Test-case-based Navigation

As part of the maintenance task, developers need to understand the *existing code* before start changing the necessary parts. In our context, the *existing code* is a piece of code already pushed to the repository and processed by the first two steps of the technique, and so it already has its annotated CG and set of association rules available as artifacts. While navigating in the *existing code*, the Test-case-based CG can assist developers in four different ways, described as follows.

- **Confirming single choice method calls:** Single choice method calls are those that do not have multiple target nodes since their names are unique through all the code. Even though source code navigation is straightforward when a method has only one implementation in the source code, dynamic languages allows creation of methods and classes dynamically at runtime. At this way, it is possible that a method call statically analyzed with one target node has multiple target methods at runtime. And so, the annotated CG assists developers confirming the target methods of single choice methods.
- **Identifying collisions by coincidence:** Call graphs are used to implement source code navigation functionalities; however, in dynamic languages the navigability precision is compromised by the fact that several edges in the CG are wrong due to type issues of dynamic languages. At this way, while navigating in the source code, several possible target nodes are shown as target candidates in a list, even though their existence does not make sense at a specific context. An annotated CG improves the suggestion list adding an extra information about test runtime execution of each item in the list, which allows developers to understand the usage frequency of each target node in the test environment. Moreover, the usage information sorts the suggestion list in a meaningful order, showing as first suggestion the most called target node in the test environment. The sorting by usage helps identifying *collisions by coincidence*, since it puts into the end of the list target methods that do not make sense in a specific context, and therefore do not have occurrences in test environments.
- **Identifying software paths uncovered by tests:** A method call with a zeroed edge counter in the annotated CG could be a case of an inefficient test coverage of the application. In this case, the annotated CG helps developers to improve the test coverage of the system, making explicit which software paths are not covered by tests. Existing

coverage tools have a different approach to handle the same issue, that is, to improve the application test coverage. These tools are used by developers seeking for coverage improvements through the use of coverage tools that generate reports about test coverage; however, these tools are not part of a developer routine. In turn, the usage of an annotated CG to build IDE functionalities puts the test coverage improvement in evidence in the developer regular workflow, since it is used by IDEs to annotate a popular IDE functionality, *Go to declaration*. In other words, while using a common tool to navigate in the code, the developer gets tips about methods uncovered by the tests.

- **Identifying dead code:** Dead code is a piece of code not used or even unreachable in the source code. In dynamic languages, it is hard to delete dead code since there is no static types to assess whether is is being used. An annotated CG helps deleting dead code since it makes visible nodes in the graph that does not have any method call as source and target methods.

Despite the fact that the Test-case-based CG assists developers in different manners, the source code navigation of methods that present *collisions by design decision* are not handled by our approach since we do not propose a solution for a problem that also exists for typed languages. Moreover, the source code navigation through *existing code* does not assist developers when changing the code, and so association rules should be used to provide guidance to this task.

4.3.2 Association Rules Navigation

Since source code modifications made by a developer in local machines are not covered by the test-case-based navigation until being processed by the first two technique steps, the set of association rules created in the second step of the technique can be used to guide developers in source code navigation.

When changing the source code, a developer can modify the CG structure in different ways, some of these type of changes are covered by the set of association rules generated in the second step. The possible modifications are: (i) add or remove a method (node); and (ii) add or remove a method call (edge). Removing an edge or a node does not impact the source code navigation negatively since it is not possible to create a multiple target methods while removing elements in the graph. In other words, based on an annotated CG, the more elements are removed from the CG, the easier the source code navigation tends to be.

In contrast, when adding a method call to the source code that has as target node a poly-

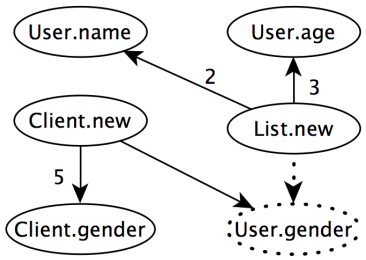

Figure 4.3 – New polymorphic method.

morphic method, it is necessary to use association rule analysis to identify the best candidate target node. Figure 4.2 shows an example of a method call addition that has multiple target methods. In the example, the method `List.new` calls a method called `gender`, but there are two different implementations of this method in the showed CG, and so it is necessary to apply association rules to understand which one of the possible target nodes fits better to the new method call. The association rules helps identify the best target node candidate based on the comparison with other usages of this method. For instance, let `X` be a method in the code (not shown in this CG) that uses the methods `User.name` and `User.gender`. The association rules allows to understand that the method `List.new` has a similarity with the method `X`, and so it can suggest that the node `User.gender` is a better candidate than `Client.gender`. At this way, association rules can be used by IDEs to suggest better target nodes when navigating in the source code.

The association rules analysis to make inferences about possible target nodes requires that existing tests cover this method, and consequently there is an annotated edge in the graph. Because of this requirement, the method (node) addition remains unhandled by both the association rule navigation technique and by the test-case-based navigation. For instance, Figure 4.3 shows a new method added in the annotated CG that creates a collision and, consequently, the need for analysis to define target nodes. Since this new code was not processed by the first two steps of the technique, the `gender` method is not considered polymorphic yet, and so there is no information about how other classes interact with it. Moreover, even if the `gender` method would be considered polymorphic, it is not possible to use the existing association rules to make inferences about the target node, since the desired target node did not exist before, and there are not rules generated that has this node as a target. Therefore, the insertion of a new method that collides with other methods is not handled by any of the two proposed source code navigation techniques, i.e. we are not able to deal with unknown unknowns. However, a workaround that could be used is that, if a method is being inserted in the code, it should be the first suggestion

in the list because there is higher probability that it will be used — this would leverage temporal aspects of code insertion.

5 EVALUATION

Our technique was developed and tuned using the previously introduced FatFreeCRM application. In order to evaluate the effectiveness of our approach, we conducted an experiment using as target system an industrial application written in Ruby. The experiment was conducted following the principles of experimental software engineering (WOHLIN et al., 2012), and our experiment goal is described using the GQM template (BASILI; SELBY; HUTCHENS, 1986). The target application has 80k lines of code and is composed of 258 classes and 1,777 methods. It is maintained by a team of 8 developers of a global consultancy company, and its goal is to assist staffing managers in the task of allocating people in projects. The system (hereafter, referred to as Staffing system) is 6 years old and was built by different teams across the globe, which resulted in a heterogeneous code base to analyze, since it includes different coding styles, ranging from design to testing aspects. The Staffing system has a broad test coverage based on unit, integration and acceptance tests, which makes it a candidate to be used as the subject of this evaluation. No further details of this application are given due to a confidentiality agreement.

Our experiment goal is to evaluate the usage of test runtime information do build CGs for dynamic languages, described in Table 5.1. Also, Figure 5.1 shows a diagram with the Goal, Questions and Metrics that guided the experiment.

Table 5.1 – Goal Definition.

Element	**Our experiment goal**
Motivation	Analyse test-case-based call graph
Purpose	evaluation
Object	with respect to precision
Perspective	from a perspective of the researcher
Scope	in the context of a large-size real-world application

To achieve the experiment goal our evaluation aims to answer three research questions (RQs):

RQ #1 *- How often source code navigation tools present multiple target nodes?* To answer this question, a procedure of data collection with developers of Staffing System aim to extract data about source code navigation in their regular routine. Three different metrics are collected:

- **Number of hours maintaining the system** - the amount of hours expended maintaining the code during the experiment.
- **Number of developers maintaining the code** - the amount of participants in the experiment.

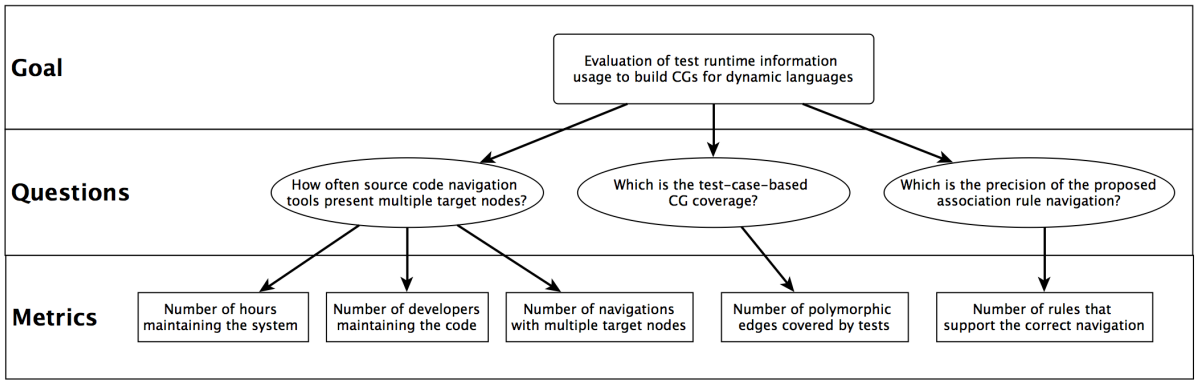

Figure 5.1 – Goal Questions Metrics Diagram

- **Number of navigations with multiple target nodes** - number of times that the source code navigation tool suggests more than one method as a target node of a method call.

RQ #2 - *Which is the test-case-based CG coverage?* Based in the test-case-based CG created for the Staffing System, manually evaluate each one of the existent polymorphic calls and its coverage by tests. To answer this question, the following metric is collected.

- **Number of polymorphic edges covered by tests** - For each polymorphic method covered by test, count the number of edges that could be removed from the graph based in a test runtime information.

RQ #2 - *Which is the precision of the proposed association rule navigation?* Based in a set of association rules generated from the test-case-based CG, evaluate how many rules derive the correct target node of a specific polymorphic method call.

- **Number of rules that support the correct navigation** - amount of rules that derive the correct target node for a polymorphic method call.

We next answer these questions providing the procedures, participants and metrics involved in the experiment, after having applied our technique to the the Staffing System.

5.1 RQ#1: Navigability Issues

In order to understand the need for a better code navigation tool for dynamic languages, it is necessary to understand the impact that this kind of tool has in the developer routine. To answer this question, a source code editor was modified to be able to trace and log code navigation in an industrial environment. The Atom Editor[1] was chosen due to its extensibility

[1]https://atom.io

and similarity with other tools used by the Staffing development team. In the Atom editor, the source code navigability tool is implemented with the popular index tool called Exuberant CTAGS[2]. Similarly to the conservative approach proposed in the *Step #1* of our technique, this tool uses method names to choose between target nodes during source code navigation.

To perform the analysis of how often navigation tools present multiple target nodes, a pair of developers coded with the modified editor in their regular routine to maintain and evolve the application during 8 days. The average time expended with the experiment was 3 hours/day. To avoid biased results, we used randomness to collect data, so there was rotation in the monitored developers, and therefore different parts of the code were tracked by the editor and developers habit particularities were avoided. The experiment generated 152 source code navigations where 19.72% were polymorphic calls, i.e. calls with multiple target node candidates. In the set of polymorphic calls, in only 36% of them the target method was shown as the first candidate in the suggestion list. In also 36% of the cases, it was shown in the second position, while in the remainder 28% of the cases it was shown after the fifth position, often after the 25^{th} position. These results emphasize the need for better source code navigation tools.

Despite the fact that CTAGS is used by many ruby developers in different source code editors, the CTAGS precision is lower than that of other competitors. For instance, Rubymine[3] presents better source code navigation suggestions for the same data collected during the experiment. However, since it is an IDE solution focused on Ruby on Rails development, ruby specific analysis and some kind of online static typing are implemented. Since Rubymine is a proprietary solution there is no much details about how the techniques are implemented and so, it is not straightforward to compare our proposed solution with it. Moreover, due to ruby oriented specific techniques, probably it is not possible to extend its approaches to other dynamic languages.

Other alternatives similar to Rubymine exist, such as Aptana RadRails[4] and Dynamic Languages Toolkit (DLTK)[5]; however solutions based on online static source code analysis are time consuming, which negatively interferes in the developer workflow. Our proposed technique, instead, aims to provide similar support without having performance issues.

[2]http://ctags.sourceforge.net/
[3]https://www.jetbrains.com/ruby/
[4]http://www.aptana.com/products/radrails/download.html
[5]http://www.eclipse.org/dltk/

5.2 RQ#2: Test Coverage of Polymorphic Calls

Our hypothesis is that a test-case-based CG can assist developers on source code navigation decreasing the number of target node candidates when the test environment exercises method calls that have multiple targets. And, to confirm the usefulness of the proposed approach, it is necessary to analyze the test coverage of our subject system. As said, the Staffing system has 1,777 methods (nodes), from which 129 has name collision, which results in 1,252 distinct method names in the source code. Despite the fact that only 7% of methods have name collision, 72% of the edges in the CG are conservative edges inserted as a result of method calls with multiple target nodes candidates due to method name collision. In this way, test coverage information can potentially assist developers by decreasing the number of target candidates in the generated CG based on test runtime execution.

The generated test-case-based CG of the Staffing system does not contain all the polymorphic methods found in the source code, being 69% of the 129 polymorphic methods present in graph. Some methods are not present in the graph since they are methods called by external libraries or by the *View* module. External libraries often require a particular interface to be implemented by the application and, therefore, it is possible to have method implementations that are only invoked by external libraries, which makes unnecessary to have navigation assistance on these cases since no other method calls will target this particular method. Typically, in the MVC architecture (adopted in the Staffing System), methods in the *Controller* module are called solely by an external library. Since *Controller* methods defines the application entry points from the *View* to *Models* and to other modules, they are only explicitly called by the adopted framework or by the *View* module. Moreover, the *View* module is not composed of classes and, consequently, there is no method definition in the code files that comprise this particular module. For this reason, method calls made from the *View* module are not present in the proposed CG. Despite the fact that source code navigation from the *View* module assists developers on source code maintainability, this kind of method call is not supported by the proposed solution since these method calls are Ruby on Rails framework specific scenarios, and this evaluation aims to understand the usage of a test-case-based CG without considering languages or framework specificities.

Without taking into account methods created for external purposes and framework particularities, 90 polymorphic methods are present in the CG. However, in the analyzed subject system, only 36% of the polymorphic methods were covered by tests runtime information of method calls. From the existent 90 polymorphic methods, 33 of them are covered by tests and

can be used to assist on source code navigation. These 33 methods are target nodes of 443 conservative edges added in the CG, and the test runtime information removed 383 conservative edges in the graph. In other words, based on test coverage, 86.4% of conservative edges added to the CG were able to be marked as not as important as calls with test coverage, having a lower position in the suggestion list when compared with method calls covered by tests. This situation leads us to two possibilities: either there are paths uncovered by test cases or there are cases of name collision. Based in a manual inspection of these 383 removed edges, only 22.9% are edges that miss test execution, while 77% of them can be removed from the CG since they were conservatively added due to method name collision. Consequently, the fact that a method call covered by tests reduced considerably the number of conservative edges encourages the usage of the test-case-based CG to assist developers on source code navigability.

5.3 RQ#3: Precision of Association Rules

To evaluate the usage of association rules to guide developers in maintenance tasks, it is necessary to understand if the last test-case-based CG built has enough information to build an online CG based on association rules. To perform the evaluation, the Staffing system source code repository was mined to identify commits that added method calls to polymorphic methods. We focused on polymorphic methods since they are difficult to analyze; simple method calls are straightforward to insert in a CG. For each commit c at a time t_i that inserts a method call targeting a polymorphic method, a test-case-based CG was created based on the commit at the time t_{i-1}. Based in the test-case-based CG constructed in the commit at t_{i-1}, a set of association rules was generated to evaluate whether the rules suggest the edge inserted in the commit at t_i. In this way, the source code repository was used as an oracle to evaluate if the modification inserted in the commit c could be derived from association rules generated based on the previous commit.

The Staffing System was rolled back to 6 points in the source code history where method calls to polymorphic methods were made. Since it is hard to install an application in a certain time in the past without an application dependency manager, it was not possible to analyze more commits for our subject. For half of commit candidates, the test-case-based CG generated did not present test runtime information about the modification inserted in the commit c, and for this reason it was not possible to generate association rules for these commits. Even though these commits present test coverage for the target methods the usage of test *doubles* masked the real method call not allowing the proposed technique to populate the CG with test runtime

information.

For the other 3 commits ($c1$, $c2$ and $c3$), association rules were created based on test runtime information and generated with the Apriori algorithm configured as follow: 0.05 of lower bound minimum support, minimum of 0.9 as confidence metric type, mining class association rules and a number maximum of rules of 100,000. To evaluate the set of generated rules, the class usage in the commit at t_i was cross checked with the class usage proposed in the rules. For example, lets say that a class `Z` that calls a polymorphic method `m()` at t_i also calls methods of classes `X` and `Y`; an association rule generated for the commit at t_{i-1} will be considered good if it holds true for the class usage of `X` and/or `Y`. At this way, association rules generated at t_{i-i} are useful to assist developers on navigation tasks if they include in the rules a suggestion to the classes that `Z` uses.

In the commit $c1$ a method call to a polymorphic method with two implementations due to collision by coincidence is introduced. Assuming that implementation A is the right one while implementation B is wrong, two datasets were created, one for each method implementation. The dataset for A generated 69,771 rules wherein 42,400 (60.77%) of them supported the modification made by the commit $c1$. That is, rules that derive the correct implementation and that in the left side hold true for some of classes that A uses. In turn, despite the fact that the dataset for the implementation B generated 100,000 rules none of them supported the desired method call. In other words, for the commit $c1$ the association rules were able to differentiate implementations with collision by coincide while 60.7% of rules generated for the right implementation supported the desired method call.

Commits $c2$ and $c3$ presents a scenario where a change in a polymorphic method with collision by design decision happens due to a method rename. Two polymorphic methods previously called `frozen?` were renamed to `hired_office?`. These methods are an example of collision by design decision, as both methods have a similar implicit interface. Each commit, $c1$ and $c2$, renamed one of the existent implementations. Since each commit only inserts a method renaming, without changing the source code, the CG structure and association rules for commit at t_i and t_{i-1} are equal. At this way, the set of association rules generated at t_{i-1} fits perfectly in the modification added by commit at t_i. In fact, $c1$ generated 3,510 rules wherein 3,110 (88.6%) supported the desired implementation. Also, the commit $c2$ generated 100,000 rules wherein 69,920 (69.9%) of them supported the desired implementation. This scenario shows that even on method name renaming tasks association rules are able to assist developers; which is not true for other approaches that are purely based on source code analysis, since after a method renaming the method call to `frozen?` does not target the class that implements the renamed

`hired_office?` method.

5.4 Threats to Validity

The main threats to validity of this dissertation are *short duration of the experiment*, *one single subject*, *lack of controlled experiment* and *manual inspection of CG coverage*.

5.4.1 Short duration of the experiment

This threat is only related to RQ#1: *how often source code navigation tools present multiple target nodes?*

The short duration of the experiment (8 days) is a threat to an internal validity, since the amount code covered by the experiment could not be representative. Method calls with multiple target methods exists in the whole subject's source code; however the data collection phase is only get data about the touched parts of the code. At this way, the short duration of the experiment is a threat to internal validity since it decreases the portion of the code covered by the experiment.

5.4.2 One single subject

The usage of one single subject is a threat to external validity, since it difficult to generalize the results to settings outside the study. To diminish this threat, it was not considered in the experiment language, domain or framework specific concepts. Moreover, experiment tasks like profiling test execution, call graph construction and association rules creation were based on well know approaches in software engineering. However, despite the fact that general approaches were chosen to make this experiment replicable, the usage of a single subject is still a threat to external validity.

5.4.3 Lack of controlled experiment

This threat is only related to RQ#1: *how often source code navigation tools present multiple target nodes?*

Despite the fact the experiment was conducted in the usual office environment with experienced professionals (around 3 years of experience), the lack of a controlled experiment increases the threat to internal validity. For instance, the nature of the task to be performed in the source

code impacts the number of times that code navigation tools are used by the developers. As an example, since the communication and dependencies between classes and methods of an old code is not fresh in developers mind, old code is often more difficult to maintain and evolve than newer code. This fact probably increases the need for source code navigation tools for old code; however it was not possible to validate this assumption since it is lacked a controlled experiment.

5.4.4 Manual inspection of CG coverage

This threat is only related to RQ#2: *which is the test-case-based CG coverage?*

The manual inspection to assert about the call graph coverage is a threat to validity since manual verifications are error prone which can lead to a misclassification of edges in the test-case-based CG. In a nutshell, the Step #1 of the proposed technique added several conservatives edges in the call graph, while the Step #2 enhanced the graph excluding some conservative edges; however, the process to assert about the set of removed edges was manually executed. Despite the fact the manual inspection increases a threat to validity, the manual inspection of the call graph was conducted by someone with knowledge in the system, which diminish the probability of misclassification.

6 CONCLUSION

Evolving enterprise software systems is one of the most challenging activities of the software development process. The usage of dynamic languages to build these systems has led to a demand for tools that support and assist developers in software maintenance and evolution tasks. However, the dynamic nature of these languages difficult the usage of traditional IDE's functionalities, since most of them are based on type definitions that are not always explicit in dynamically typed languages. *Go to declaration* is an example of an IDE tool, based on a Call Graph structure, that assist developers and that has its performance affected by the type definition absence. Different approaches have been proposed to deal with this problem, like static type inference, dynamic analysis, conservatively approximated call graphs; however dynamic languages still lack advanced tooling support with efficiency and precision to assist developers.

In this work, we proposed a hybrid approach (static and dynamic, online and offline) based on steps to build a test-case-based call graph for dynamic languages. Step #1 builds a conservative static call graph and profile test execution to extract information about runtime method calls. Based on test runtime information, Step #2 enhances the static call graph labeling actual method calls (made on runtime) and generates a set of association rules. The first two steps ran offline and its outcomes, test-case-based CG and association rules, are used by Step #3 to assist developers on source code navigation.

The proposed technique was evaluated in an industrial application written in Ruby. The test-case-based CG showed a significant improvement in the CG precision by removing 77% of conservative edges. Moreover, the association rules were evaluated against historical information stored in source code repository and proved to be able to suggest proper target nodes when multiple targets are available. Also, the association rules allowed to distinguish method *collisions by coincidence* and *design decision*.

The main contribution of this work is the creation and evaluation of a test-case-based call graph. The usually ignored dynamic information of test executions showed effectiveness to enhance the construction of call graphs for Ruby language. The applicability of test-case-base CG to other dynamic languages should be evaluated and is not part of this work. However, the usage of general and well know approaches in software engineering, like test profiling, call graph construction and association rules creation, avoiding any language or framework particularities indicates that our proposed technique is replicable to other environments.

Despite the fact that the proposed technique showed good results when evaluated against a real-world application, it is worth mention that some requirements should be fulfilled to allow

its usage. Since the technique depends heavily on test scenario executions, to take advantage of our technique, an application should present a consistent and automated test environment. We consider the following characteristics as a consistent test environment: *A. Broad test coverage* - since we propose a path-sensitive analysis based on test runtime, the broader the test coverage, the better is the outcome. For this reason, an application should have a broad test environment to apply the proposed technique. However, our technique can guide developers on creating missing test scenarios pointing out uncovered paths in the system, at this way, it is possible to minimize the problem of not having a broad test coverage; *B Pipeline configuration* - since we propose a technique that relies on offline executions on pipeline machines, it requires that developers have access and permission to configure pipeline machines to make them able to profile runtime executions.

Moreover, our technique does not solve the problem of deciding target nodes of polymorphic method calls when target nodes collide by design decision. As well as it happens in statically typed languages, it is not always possible to decide target nodes when some collision by *design decision* are used, Java *interfaces* for instance. At this way, the proposed technique aims at discovering target methods for method calls that *collide by coincidence*, although it also presents test runtime information about any type of collision.

6.1 Future Work

Test good practices encourage developers to create tested systems that follow a test pyramid principle, which is fine-grained tests at the basis of the pyramid while end-to-end and cross functional test at the top. At this way, the most relevant tests to our proposed technique are in the middle or top of the pyramid, since they not mock method calls as unit tests usually does. Hence, as future work we plan to consider test double definitions when constructing the test-case-based CG since several fine-grained method executions were not considered in the proposed technique. Despite the fact that test double definitions does not represent actual method calls (since they fake behavior), it is possible to implement a ternary analysis where an edge in the call graph can be considered: covered by test execution, covered by test double, not covered.

The proposed technique consist in both static and dynamic analysis. However, the static analysis (Step #1) does not innovate in any aspect, since it only uses a conservative and field-based technique. At this way, the usage of a more precise technique in the creation of a static call graph should improve the overall results when combining static and dynamic analysis. Finally,

we have an assumption that the usage of test runtime information in developers routine eases the comprehension of test environment improving its coverage.

REFERENCES

AGRAWAL, R.; IMIELIŃSKI, T.; SWAMI, A. Mining association rules between sets of items in large databases. In: ACM. **ACM SIGMOD Record**. [S.l.], 1993. v. 22, n. 2, p. 207–216.

AGRAWAL, R. et al. Fast discovery of association rules. **Advances in knowledge discovery and data mining**, AAAI/MIT Press Menlo Park, CA, v. 12, n. 1, p. 307–328, 1996.

AN, J.-h. D. et al. Dynamic inference of static types for ruby. **ACM SIGPLAN Notices**, v. 46, n. 1, p. 459, jan. 2011. ISSN 03621340. Available from Internet: <http://portal.acm.org/citation.cfm?doid=1925844.1926437>.

ANDERSON, C.; GIANNINI, P.; DROSSOPOULOU, S. Towards type inference for javascript. In: **ECOOP 2005-Object-Oriented Programming**. [S.l.]: Springer, 2005. p. 428–452.

BASILI, V. R.; SELBY, R. W.; HUTCHENS, D. H. Experimentation in software engineering. **Software Engineering, IEEE Transactions on**, IEEE, n. 7, p. 733–743, 1986.

BECK, K. **Test-driven development: by example**. [S.l.]: Addison-Wesley Professional, 2003.

BOHNER, S. A. Software change impact analysis. Citeseer, 1996.

CANNON, B. **Localized type inference of atomic types in python**. Thesis (PhD) — CALIFORNIA POLYTECHNIC STATE UNIVERSITY San Luis Obispo, 2005.

FELDTHAUS, A. et al. Efficient construction of approximate call graphs for javascript ide services. In: IEEE PRESS. **Proceedings of the 2013 International Conference on Software Engineering**. [S.l.], 2013. p. 752–761.

FOWLER, M.; FOEMMEL, M. Continuous integration. **Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf**, 2006.

FURR, M.; AN, J.-h. D.; FOSTER, J. S. Profile-guided static typing for dynamic scripting languages. **ACM SIGPLAN Notices**, ACM, v. 44, n. 10, p. 283–300, 2009.

FURR, M. et al. Static type inference for ruby. In: ACM. **Proceedings of the 2009 ACM symposium on Applied Computing**. [S.l.], 2009. p. 1859–1866.

GAMMA, E. et al. **Design patterns: elements of reusable object-oriented software**. [S.l.]: Pearson Education, 1994.

GROVE, D.; CHAMBERS, C. A framework for call graph construction algorithms. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM, v. 23, n. 6, p. 685–746, 2001.

HEINTZE, N.; TARDIEU, O. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. In: ACM. **ACM SIGPLAN Notices**. [S.l.], 2001. v. 36, n. 5, p. 254–263.

JANG, D.; CHOE, K.-M. Points-to analysis for javascript. In: ACM. **Proceedings of the 2009 ACM symposium on Applied Computing**. [S.l.], 2009. p. 1930–1937.

LHOTÁK, O.; HENDREN, L. Scaling java points-to analysis using spark. In: SPRINGER. **Compiler Construction**. [S.l.], 2003. p. 153–169.

MENS, T. et al. Challenges in software evolution. In: IEEE. **Principles of Software Evolution, Eighth International Workshop on**. [S.l.], 2005. p. 13–22.

MESZAROS, G. **xUnit test patterns: Refactoring test code**. [S.l.]: Pearson Education, 2007.

RYDER, B. G. Constructing the call graph of a program. **Software Engineering, IEEE Transactions on**, IEEE, n. 3, p. 216–226, 1979.

TIOBE Software Corporation. <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Accessed: 2015-08-10.

TOMA, T. R. et al. A dependency graph generation process for client-side web applications. 2015.

TRATT, L. Dynamically typed languages. **Advances in Computers**, Elsevier, v. 77, p. 149–184, 2009.

WHITE, S. A. Introduction to bpmn. **IBM Cooperation**, v. 2, n. 0, p. 0, 2004.

WOHLIN, C. et al. **Experimentation in software engineering**. [S.l.]: Springer Science & Business Media, 2012.