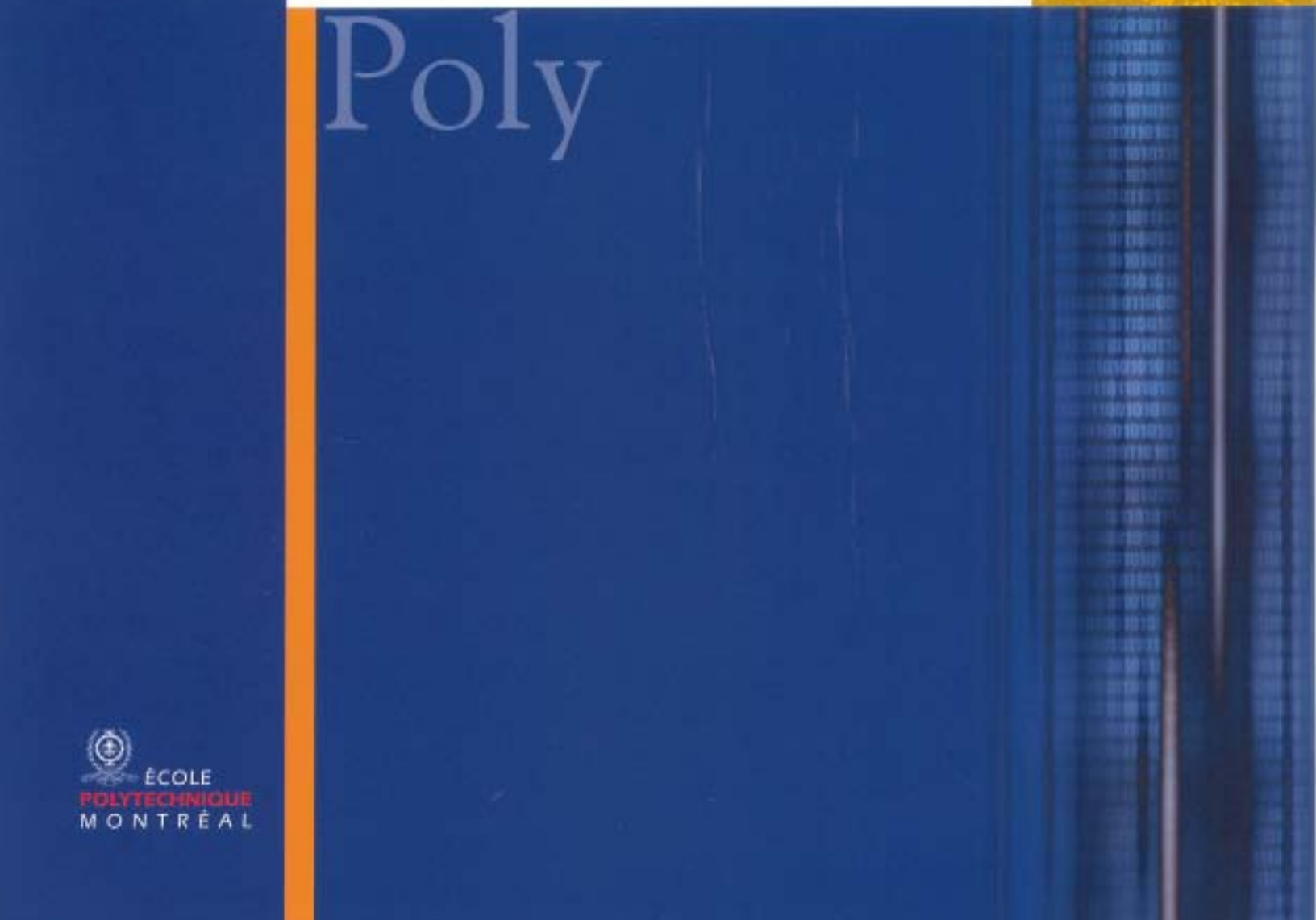
EPM–RT–2010-06

CHANGE IMPACT ANALYSIS OF MULTI-LANGUAGE AND HETEROGENEOUSLY-LICENSED SOFTWARE

Ferdaous Boughanmi
Département de Génie informatique et génie logiciel
École Polytechnique de Montréal

Septembre 2010



ÉCOLE
POLYTECHNIQUE
MONTRÉAL

EPM-RT-2010-06

Change Impact analysis of Multi-Language and Heterogeneously-licensed Software

Ferdaous Boughanmi
Département de génie informatique et génie logiciel
École Polytechnique de Montréal

Septembre 2010

Dépôt légal :
Bibliothèque nationale du Québec, 2010
Bibliothèque nationale du Canada, 2010

EPM-RT-2010-06
Change Impact analysis of Multi-Language and Heterogeneously-licensed Software
par : Ferdaous Boughanmi
Département de génie informatique et génie logiciel
École Polytechnique de Montréal

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Change Impact analysis of Multi-Language and Heterogeneously-licensed Software

by

Ferdaous Boughanmi

A proposal submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Département de génie informatique et génie logiciel

September 2010

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Abstract

Département de génie informatique et génie logiciel

Doctor of Philosophy

by Ferdaous Boughanmi

Today software systems are built with heterogeneous languages such as Java, C, C++, XML, Perl or Python just to name a few. This introduces new challenges both for the software analysis domain and program evolution as programmers must to cope with a variety of programming paradigms and languages. We believe that there is the need of global views supporting developers to effectively cope with complexity and to facilitate program comprehension and analysis of such heterogeneous systems. Furthermore, the heterogeneity of the systems is not limited to the language but also impacts the components licensing. In fact, licensing is another type of heterogeneity introduced by the large reuse of open source code. The heterogeneity of licenses also introduces challenges such how to legally combine components in different programming languages and licenses in the same system and how the change of the software can create a violation of licenses. In this context, we would like to develop a re-engineering tool for analysing change impact of heterogeneously licensed system considering multi-language environment. First, we want to study change impact analysis in multi-language system in general and extend it to support the issue of licenses.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

During the software life cycle, about 52% of effort is spent on maintenance and 47% of this effort is estimated to be devoted to program understanding [1]. What makes maintenance more challenging is the heterogeneity of programming languages and paradigms, in fact large software application are composed of modules often coded with different programming and specification languages. Maintainers thus face a variety of different programming languages making program comprehension and software maintenance difficult and expensive.

We believe that there is a need for tools to support the analysis of large heterogeneous systems. Indeed, Most re-engineering tools focus on extracting dependencies from a single programming language, e.g., extracting the call graph from a C program but they are hardly able to deal with two or more programming languages. Furthermore, they do not handle multi-language programs as a single entity, because they do not manage the interconnections between different parts of code written in different languages. For example, the call graph is produced for each program written in one language independently from the others.

Multi-language programs often have also heterogeneous licenses. Heterogeneous licensing is the consequence of the availability of Open Source Software (OSS) and also proprietary system with open APIs. Developing system by reusing existing components decreases the cost of developing phase. Yet, it can introduce another type of problems: due to the various rights and obligations of each license and the large number of licenses and their different versions that can be conflicting, it is difficult to respect all obligations. In addition, software engineers are not not well trained in the legal issues are faced with a complex array of legal rights and obligations that they have difficulty to track and

understand.

In this context of heterogeneous programs, our goal is to develop a re-engineering tool for analysing change impact in heterogeneously licensed and multi-language systems. First, we want to study impact of the change in multi-language systems in general and extend it to support the issue of licenses. We want to manage the evolution of the system by providing the impact of change, for example architecture evolution like modification of the interconnection between two modules or license evolution (update to a new version) can introduce inter-license conflict. Also, we want to suggest possible architectures to avoid legal conflict between licenses by using system expert.

1.1 Context

1.1.1 Why we develop multi-language system and heterogeneously-licensed software?

- Programming Pearls, Communications of the ACM, Sept. 1985

"Scripting is almost always a more pleasant and productive alternative to using a systems programming language. Scripting languages are not designed to do everything, however, and there comes a time when you need to dig down to C/C++ for speed, fine-grained data structures, type safety, and access to existing libraries. The ability of languages such as Perl, Visual Basic, Python, and Tcl to integrate well with C accords them the status of a serious development language, in contrast to awk and early versions of BASIC, which were seldom used for production applications."[2].

Other than the increasing productivity to develop multi-language systems, we distinguish many other reasons:

- Efficiency:
  for performance reason, a high level language (e.g, Java) may invoke code written in low level language (e.g. C), for example to access to material layer.
- Suitability:
  a programming language is characterized by a set of language features. These features influence how the programming languages is used, which contexts it is best used in, and for what purpose. For example, if we use Python to develop a software

and access to explicit implementation details, such as, memory management is needed, then it will be more suitable to use a language that provides this features (e.g, C).

- Reuse:
  the availability of Open Source Software encourages software developers to reuse them. Consequently, developers prefer to develop the glue code.

1.1.2 Open Source Software

Open source software (OSS) development has some typical characteristics, such the widespread reuse of components and licenses. This widespread of various and different licenses increases the difficulty to understand their constraints. Consequently, new re-engineering tool must consider the licenses analysis. OSS development process outputs have been studied to study many aspects of programs, for example in [3], they analyzed a sample of around 400 projects from a popular OS project repository. Each project is characterized by a number of attributes. According this study, the most used languages were C, C++, Perl, and Java. Thus, we are interested to study these languages. Despite the large number of OSS projects, developments effort have focused on a few large projects such as Linux, Mozilla, and Apache. In [3], Capiluppi and al. confirmed that few projects are capable of attracting a meaningful community of developers. The majority of projects is made by few (in many cases one) person with a very slow pace of evolution. We think that the analysis of licenses will be more useful in project with great community and in constant evolution because the evolution of the systems increase the threat of license violation and the large number of components and licenses increases the constraints to respect inter-licenses compatibility. Hence, we will apply our study to Fedora-12 (Linux distribution).

1.1.2.1 Collective and derivative works

Distinguishing between collective work and derivative work is fundamental for analysis of legal issues of components based software system.

A collective work is:

A work in which a number of contributions, constitut- ing separate and independent works in themselves, are as- sembled into a collective whole. (17 U.S.C. 101.)

And a derivative work is:

A work based upon one or more preexisting works, such as a translation or any other form in which a work may be re- cast, transformed, or adapted. (17 U.S.C. 101.)

1.1.2.2 Types of Licenses

Licenses can be categorised into four categories:

1. Academic Licenses, *"so named because such licenses were originally created by academic institutions to distribute their software to the public, allow the software to be used for any purpose what so ever with no obligation on the part of the licensee to distribute the source code of derivative works. The Berkeley Software Distribution (BSD) license used by the University of California to distribute its software is the archetypal academic license. Academic licenses create a public commons of free software, and anyone can take such software for any purposeincluding for creating proprietary collective and derivative workswithout having to add anything back to that commons."* [4]

2. Reciprocal Licenses, *"Allow software to be used for any purpose whatsoever, but they require the distributors of derivative works to distribute those works under the same license, including the requirement that the source code of those derivative works be published. The GPL license, written by Richard Stallman and Eben Moglen at the Free Software Foundation, is the archetypal reciprocal license. Anyone who creates and distributes a derivative work of a work licensed under a reciprocal license must, in turn, license that derivative work under the same license. Reciprocal licenses, like academic licenses, contribute software into a public commons of free software, but they mandate that derivative work also be placed in that same commons."* [4]

3. Standards Licenses, *"are designed primarily for ensuring that industry standard software and documentation be available to all for implementation of standard products. These licenses sometimes require that any differences from the industry standard be published as a reference implementation so that the standard may evolve if necessary."* [4]

4. Content Licenses, *"ensure that copyrightable subject matter other than software, such as music, art, film, literary works, and the like, be available to all for any purpose whatsoever. These licenses are discussed more fully on the Creative Commons website at* `www.creativecommons.org`. *While the Creative Commons goals are*

not directly related to software freedom, there are many similarities of objective. A few of the software licenses discussed in this book, in particular the Academic Free License (AFL) and the Open Software License (OSL), are appropriate for use with content as well as software, as will be explained in due course." [4]

1.1.2.3 GPL, BSD, Apache

We now present the most used licenses [5]: GPL, BSD, and Apache, which we will use in our project to provide the possibles architectures of heterogeneously licensed system.

1. BSD: Academic License

   Contrary to the GPL License, BSD allows anyone to redistribute the work or any derivative work without any source, if such is the desired path. So BSD do not cause incompatibility problem : the caller of program under BSD license can use any license.

2. GPL: Reciprocal License

   GNU Public License, it is very common license for open source packages. GPL is known for having strict reuse constraints. So it is important to focus on incompatibility issues involving GPL license.
   GPL is reciprocal license because any software that reuses code licensed under GPL should be licensed under the same version of the GPL. Here the GPL say it:

   "You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this license."(GPL, Section2) They are strong conditions on how a caller can use GPL package.

   They are strong conditions on how a caller can use GPL package. The GPL requires to analyse the software based not upon how it is linked but upon how it is distributed. *"These requirement apply to the modified work as whole. if identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this license, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the must be whole on the terms of this License,*

whose permissions for other licenses extend to the entire whole, and thus to each and every part regardless of who wrote it". (GPL section 2)

According to the first sentences, the GPL is applied to "modified work as whole". A modified work is derivative work (17 U.S.C). There is no hint that linking makes a difference. The second sentences refers to portions of the work that are not derived from the program (have their own copyright owners and their own license). A work must be independent and separate works are linked in some way to the GPL program. Such works remain "independent and separate works," at least "When you distribute them as separate works," and the GPL cannot possibly apply to them without their copyright owner's consent.

In the GPL, we must analyse the software on how it is distributed. We converted linking limitations to the interconnection type:

- if the caller uses via fork/exec then the caller can have any license.
- if the caller uses called components as a plugin then the caller can have any license.
- if the caller uses linking as types of connexion so it must be licenses under the same version of GPL.

The program licensed under academic open source licenses can be incorporated into GPL-licensed software but the converse is not true.

The are some licenses are not compatible at all with GPL, we will limit these list to the different version of the concerned licenses (BSD, Apache):

- Apache License, version 1.1
  This is a permissive non-copyleft free software license. It has a few requirements that render it incompatible with the GNU GPL, such as strong prohibitions on the use of Apache-related names.
- Apache License, version 1.0
  This is a simple, permissive non-copyleft free software license with an advertising clause. This creates practical problems like those of the original BSD license, including incompatibility with the GNU GPL.

And the compatible version of our chosen licenses (BSD, Apache) are:

- Apache License, version 2.0

  This is a free software license, compatible with version 3 of the GPL. This license is not compatible with GPL version 2, because it has some requirements that are not in the older version.

- Modified BSD license

  This is the original BSD license, modified by removal of the advertising clause. It is a simple, permissive non-copyleft free software license, compatible with the GNU GPL. If we want a simple, permissive non-copyleft free software license, the modified BSD license is a reasonable choice.

Case of plugin [6]: The legality depends on how the program invokes its plug-ins. For instance, if the program uses only simple fork and exec to invoke and communicate with plug-ins, then the plug-ins are separate programs, so the license of the plug-in makes no requirements about the main program. If the program dynamically links plug-ins, and they make function calls to each other and share data structures, we believe they form a single program, which must be treated as an extension of both the main program and the plug-ins. To use the GPL-covered plug-ins, the main program must be released under the GPL or a GPL-compatible free software license, and that the terms of the GPL must be followed when the main program is distributed for use with these plug-ins. If the program dynamically links plug-ins, but the communication between them is limited to invoking the main function of the plug-in with some options and waiting for it to return, that is a borderline case. Using shared memory to communicate with complex data structures is pretty much equivalent to dynamic linking.

3. Apache License, version 2.0: Academic license

   The Apache License is a free software license authored by the Apache Software Foundation (ASF). The Apache License requires preservation of the copyright notice and disclaimer, but it is not a copyleft license, it allows use of the source code for the development of proprietary software as well as free and open source software [7][8].

   All software produced by the ASF or any of its projects or subjects is licensed according to the terms of the Apache License. Some non-ASF software is licensed using the Apache License as well. As of July 2009, over 5,000 non-ASF projects located at SourceForge.net are available under the terms of the Apache License. In a blog post from May 2008 [9], Google mentioned that 25% of the 100,000 projects then hosted on Google Code were using the Apache License.

Like any free software license, the Apache License allows the user of the software the freedom to use the software for any purpose, to distribute it, to modify it, and to distribute modified versions of the software, under the terms of the license. The Apache License, like BSD licenses, does not require modified versions of the software to be distributed using the same license (in contrast to copyleft licenses). In every licensed file, any original copyright, patent, trademark, and attribution notices in redistributed code must be preserved (excluding notices that do not pertain to any part of the derivative works); and, in every licensed file changed, a notification must be added stating that changes have been made to that file [7][8].

1.1.3 Organization of the report

Chapter 1. Introduction: We begin by introducing the context of our project and what lead us to work on multi-language and heterogeneously licensed system. And we expressed the directions that interest us: Change impact analysis on technical aspect and extending it to legal aspect. And the second section, we explain the factors that encourages software engineers to develop a multi-language system. and we present the open source software and the related concept that we need. Finally, we present the plan of our proposal.

Chapter 2. Problematic and research questions: Then, in Chapter 2, we shall establish a list of goals and research questions and the flow between sub questions. Two directions will be distinguished in this chapter: How we can analyse the change impact in multi-language program and extending it to deal with heterogeneously licensed systems. We will details the challenge to answer each questions and the difficulties that we face.

Chapter 3. State of the art: This chapter is devoted to present the related work in the domain of re-engineering of multi-language system and heterogeneously licensed system. We presented two types of work: that relevant in the domain and that influenced our choice and approach.

Chapter 4. Approach: In chapter 4, we pose our approach to resolve our research questions. We will stress on dependencies extraction and type of interconnection that are necessary for change impact analysis and studying licenses and also formalisation of system representation that helps us to develop a system expert that will answer the

questions about the possible architectures.

Chapter 5. On Going Work: In the context of our project, we initiated two on going work. The first we want to answer on the research question: What is possible architectures that can be obtained by combining components written in C/Java/Perl and having different licenses that can be Modified BSD, Apache, or GPL. To answer this question, we investigated the interconnection types between the languages: Java, C, Perl and we formalised the representation of system including licenses that will permit to us to develop a tool using system expert. Second, we preparing a preliminary work for change impact analysis of heterogeneously licensed system. We explained how we want to extend the work of German and al. [10]. German and al. proposed a method to understand licensing compatibility issues in software packages but they identify the dependencies types manually. So we want to extend their work by automating the identification of some type of dependencies.

Chapter 6. Conclusion and Planning: We will resume our goal and the flow of the our approach to answer our research goal. And we will present our planning and the conference that we will aim during the thesis.

Chapter 7. Annexe: In the chapter on going work, we are supposed to present the possible interconnections between the languages Java, C, Perl but we presented just Java/C. So, we present the interconnection between Perl/C and Perl/Java in the annexe.

Chapter 2

Problematic and research questions

Most software are built by combining several components and they reuse existent code in diverse language. The advent of free/open source software (FOSS) has amplified this activity by providing software components that are ready for reuse. Consequently, there is a widespread of heterogeneously-licensed systems combined with multi-language aspect of their components. So this poses new challenges in two directions: the multi-language handle and licenses issue. The heterogeneity of licenses introduces threat of incompatibility of licenses that depends on the architecture of the system and how the components are interconnected and multi-language aspect introduces difficulty to analyse systems as whole such extracting the dependencies. So, such environment increases the complexity of maintenance activity [1]. For example, to modify (delete, change, add) a given method in a module, we must verify if this modification introduces errors in another module that uses this method directly or indirectly and if it violates a term of licences.

Example of multi-language program:

This is an example of multi-language program written in Perl, Java and C. It permits to display the temperature or the humidity rate depending if the user choose "temp" or "hum". The principal program asks the user to type "temp" or "hum". After that, a Java program, meteo.java, is invoked via system call. This program uses native methods printTemp() and printHum(), to display the temperature and the rate of humidity. These native methods are implemented in C and have the respective signatures: *JNIEXPORT void JNICALL Java_meteo_printTemp (JNIEnv *, jobject)* and *JNIEXPORT*

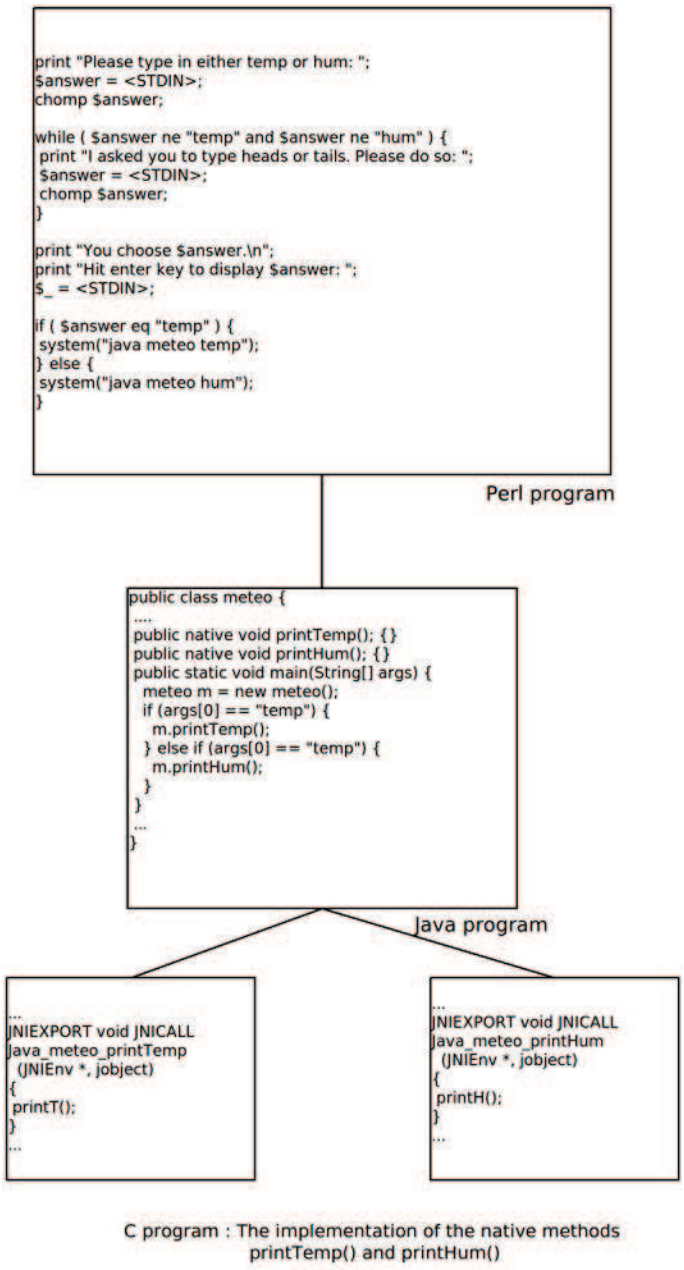


FIGURE 2.1: Example of Multi-language program using Java, Perl and C

*void JNICALL Java_meteo_printHum(JNIEnv *, jobject).*

Suppose that we modify the implementation of the native method *Java_meteo_printTemp (JNIEnv *, jobject)* by adding a new parameter corresponding to the unit of measure of the temperature. We must know the updates that we have to apply after this modification, to let the program works with the new version of the method *Java_meteo_printTemp (JNIEnv *, jobject, jint).*

The developer of such program wants to know what is the impact of a modification before accomplishing it, because it can introduce errors that are difficult to resolve.

Among the problems faced by developers during software maintenance is when a new

version of a component is available, they can't decide if they can update this component without affecting the functioning of the system. To manage the version of system components, we need the dependencies but the extraction of dependencies of the components is not easy because the dependencies can be complex. So the propagation of the update effect is not well mastered.

The goal of our work is the creation of a model to analyse a change impact of multi-language systems and to develop a tool that supports this model. Also, this model will help us to deal with heterogeneously-licensed systems because it will include data about components, their dependencies, and we want to study the impact of modification on the license compatibility using the result of previous steps.

We begin by presenting the challenge for multi-language system in general after we present the questions in the special case of heterogeneously-licensed software and how the first challenges help us to analyse this type of system.

2.1 Change impact analysis

Question 1 : How we can analyse the impact of modification of an entity (method, file,...) in multi-language system?

Change impact analysis provides the potential consequences of a change, or estimated what needs to be modified to accomplish a change [11]. Our challenge is to define a strategies to detect the effect of changes in a multi-language system. The change impact analysis permits to simplify the evaluation of change request and helps to detect incoherence, for example, when two communicating modules that exchange data and we modified the format of output data of the module that sends the data, thus the format became incompatible with the format of the input data requested by the modules that receive this data. Change impact analysis is more difficult in the case of heterogeneous system due to the complexity of dependencies between languages.

We present a case of heterogeneous system to illustrate the difficulties induced by this type of software:
Beltrame et al. presented ReSP in [12], a hardware simulation platform targeted to Multi-Processor Systems-On-Chip; the platform is based on the integration of Python and SystemC allowing effortless integration of external IPs and custom components. They use Python because it augments ReSP with the observability of the internal structure of SystemC components using the reflective capabilities. The use of Python enables

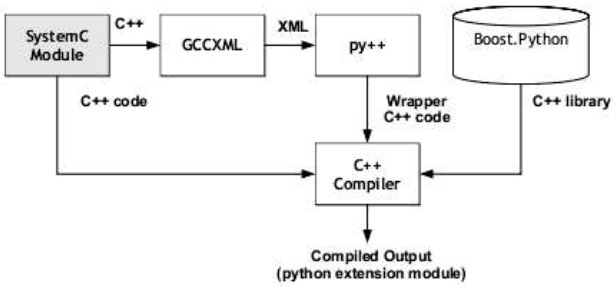


FIGURE 2.2: ReSP wrapper generation flow

a fine grained control over simulation and over the internal status of the component modules. The potentialities offered by the integration of Python and SystemC are exploited, during simulation, to query, examine, and possibly modify the internal status of the hardware models.

ReSP provides a wrapper for the Python scripting language around the SystemC kernel. Python inherently supports reflection, and allows access to SystemC variables and arbitrary function calls to SystemC. The Simulation Controller is a set of Python classes that translate commands coming from the user into SystemC function calls, controlling the simulation behaviour. The novelty introduced by ReSP lies in the Python wrapper generation for SystemC and TLM components. ReSP deals with this step automatically, by generating the Python wrapper right after parsing the component C++ header file. The generation flow is shown in Figure 2.2.

During a discussion with G. Beltrame, he affirmed that the main difficulty to manage such heterogeneous system is the managing of the tools used, for example: there is new version of a tool and he wants to update this tool but he can't be sure that he can recompile without errors and even if he can compile, then there a risk that the system will not work well like before the update. Specially, the problem arises with the tools to generate the wrapper. For, example Boost.Python 1.42 is not compatible with py++ 2.4.3. We think this difficulty is due to the missing of the tools that manage dependencies of heterogeneous system.

Change impact analysis could help to reduce the risk of change in a system. To do this analysis, we need to extract dependencies into suitable model to study the propagation of the change is necessary. This extracted data must be exploited to answer questions like: when we rename a function, what is the modules affected by this change? Or, when we want to update a library, is new version is compatible with the rest of the system?

2.1.1 Dependencies extraction

Question 2 : How we can extract effectively the dependencies in Multi-language system?

Dependencies extraction is a difficult task because there are many different programming languages, we can have files and or modules consisting of several languages. We see that it is difficult and perhaps impossible to provide parser suitable for all types of files. For example, suppose we have a parser to extract the dependencies between Java and C. This parser can only analyse a program written with these two languages, we can not for example use it to extract the dependencies in a program written in Perl and C. So it is difficult to have a generalized parser that can analyse any multi-language program.

Re-take the example of the figure 2.2, A system call in Perl program is like that:

```
system("command");
```

But, in Java the system call is done using the class *Runtime*:

```
Runtime rt = Runtime.getRuntime();
rt.exec("command");
```

Our goal is to study the possibility of generalization of the data extractor by targeting a particular paradigm or language and the type of data to extract. As software became more and more large, it is better to have a flexible parsers that give us the ability to control the data extracted and the file that we want to analyse and permits us to exclude details to lighten the extraction of data. How can we make an efficient and accurate analysis to extract the dependencies data?

2.1.2 Representation of extracted data

Question 3 : How we model the dependencies in a multi-language?

To automate the analysis of multi-languages system, we must be collect and process data from different sources. Only a system with a global view allows a correct analysis. So the key of multi-languages analysis is a common model that supports the concepts of various programming languages [13] or several models that are linked logically. We believe that an effective analysis requires that we do not do this analysis on each component independently, but take into account data on interconnections that are overlapped

between two pieces of code that must be logically connected in our model or linked models. However, such a model that supports the interconnection between different languages is not an easy task because the inter-language communication methods differ from one couple of language to another. Let's take the example of figure 2.1:
the system call made within the Perl program takes a string as a parameter, static analysis may not be effective because the command can not be resolved efficiently.

Our challenge is to design a model to represent dependencies in a multi-language system. This model will describe the entities in the program and their interconnections type. This model will serves to analyse the impact of change in a component-based program and will be extended to be used to analyse the heterogeneously licensed software.

2.2 Heterogeneity of the licenses and their constraints

The heterogeneity of components-based system does not only concerns the heterogeneity of the languages used to develop each component, but also the licenses of these components. This heterogeneity of licenses can introduce threats of legal violation of the terms of licenses. Software licenses are constraint that must be respected in the development of the software. Moreover, legal constraints impose certain architectural styles (black box, white box...) and connectors (linking, fork/exec...) between modules [14]. We want to focus our research on systems that includes components with different licenses.

Question 4 : What is the type of problem and constraint that can be introduced by the heterogeneity of licenses?

The Intellectual property(IP) are expressed in terms of the licenses, rights, and obligations. They include: the right to use, distribute, sublicense, interoperation of the system with specific IP regimes. This IP can have conflicts with other licenses'obligations. So, the combination of different licenses in a single system is not simple because each license introduces constraints on the way of use (distribution, copy,...) that can be incompatible and also how we can reuse a program by integrating it to another system or modifying it. We have to know the IP to be able to identify the possible legal combinations of licenses in one system.

For example, when programmers want to develop a system S under a license L that reuses an open-source component C, they must verify whether they respect the restrictions of the grant given by the license of C. In fact, a component can be reused to create from it

a derivative work mainly by using white-box form that permits to use one or more files of C, either in the original or modified form. It can be also used as part of collective work that is usually realized via black box form for example by calling components as executables. But determining whether a work is derivative or collective work for a black box reuse is difficult because it depends on the nature of the use and the interconnection type.
Consider the following scenario: suppose we want to distribute a system S under a proprietary license P and one of the component C_i of S is licensed under the terms of GPL_2. C is interconnected to S via black-box linking, so S is a derivative work of C. GPL_2 imposes that all derivative work S made from component under GPL_2 must be also licensed under GPL_2. In contrast, if we modify the interconnection type, and that black box forking is used instead of black-box linking, then, according to the FSF, S is not a derivative work of C. In this case GPL_2 gives grant to distribute S under a proprietary license [14][4]. This example show us how can the interconnections type be a constraint to respect the IP and it depends on the licenses used and their versions and it is complicate to verify this respect of the IP of a large software.

Question 5 : What are the possible architectures of a heterogeneously licensed system?

To combine conflicting licenses, developers must adapt and modify their technical solutions and architectures to remove this conflict. As we explained before, the suitable connectors must be chosen to integrate a component to a system in order to ensure the compatibility of licenses. So the license of a component affects the requirements, the architectures and the potential uses of a component-based application[14]. The licenses mismatch is a complex problem for which software engineers have a limited knowledge and many software organizations are warned about the incorporation of the open source component in existent system [15][16][17][18]. That's why, it is interesting for developer to have a tool that gives the possibles solutions to integrate a component legally. We would like to present possible architectures that can be obtained by combining components written in C/Java/Perl and having different licenses. Thus we want to make the developers aware that certain modification may violate licensing constraints.

Question 6: How can we identify a legal violation in system with heterogeneous licenses?

Suppose that we have a component based system. And the components are possibly written in various language and licensed under different licenses. How can we proceed to identify if there is a legal violations. We know that licenses introduce a constraint on the interconnection type. Here we have a need to the question 2: the dependencies

extraction and we have to identify in addition the type of the interconnection. Having the dependencies and the type of interconnections between the modules and the licenses of this modules and licenses of the final system and all the terms of the licenses. We must be able to know if the system respects all the terms of this licenses else we identify the violations and what are the possibles alternatives to remove this violation (question 5).

2.3 Change impact on license compatibility analysis

Question 7: What is the impact of modification on the license compatibility

When we perform a modification on software components, it can affects the legality of the system by causing licenses violations. For example, when a component is updated to a new version, the new version can have a different license than the old one, this licenses can be incompatible with the licenses in the system. Another example, we will reuse the example in the figure 2.1, suppose that the Java program meteo is licensed under GPL and the Perl program is licensed under Modified BSD and the Perl program call the Java one by system call. If we modify the type of interconnection to dynamic linking using *JPL::Class* to load meteo class, the GPL license is violated because the Perl and the Java program is considered derivative work of the Java program, so the Java program must be licensed under GPL.

Chapter 3

State of the Art

In this section, we present the main existing work in the field of re-engineering of multi-language program and the licenses mismatch problem.

3.1 Re-engineering of multi-language program

Several authors addressed issues related to multi-languages software applications. Early works such as the one of Linos [19] found that programmers organize their implementation, integration, and maintenance activity based on specific programming paradigms which justifies the additional complexity like type matching perceived when integrating components developed with different languages. The imperative and procedural language programmers focus on procedures; in contrast the object-oriented programmers, tend to focus on objects. Hence, the objective of Linos is to provide a tool that facilitates understanding, re-engineering, implementation and integration of multi-language programs. For this reason, he investigated the dependencies between programs written in different paradigms. His approach is based on the classification and formalization of the program components and their dependencies. He proposed a taxonomy of the different types of dependencies according to the paradigm of programming languages and a dependencies formalization.

- In procedural and imperative language: The author call the components and their relationships *Procedural Program Dependencies* (PPDs). A PPD can be presented as PPD $= < X, Y, R >$ Where X and Y can be data elements and data types or sub-programs and R depicts a relationship between X and Y. For example, the triplet <Variable, Type, is-defined-as> presented the is-defined as relationship between variables and data-types.

- In the functional language: The components and their relationships are called Functional Program Dependencies (FPDs). A FDP can be presented as FDP = $< X, Y, R >$ where X and Y can be atoms , expressions or lists and they are linked with the relationship R. For example, the triplet <Constant, List, is-included-in > presents the is-included-in relationship between constants and lists.

- In Object-Oriented language: The components and their relationships are called Object-Oriented Program Dependencies(OOPDs). OOPDs = $< X, Y, R >$, entities X and Y can be data-objects, class-types or methods and R represents a relationship between X and Y. For example, the triplet <Class, Method, implements> defines the implements relationship between Classes and Methods

- In Logic language: Logic Program Dependencies include logic programs elements such as facts, rules, and data-arguments and their relationships. Example of LPD = $< Fact, Rule, uses >$ which presents the uses relationship between facts and rules.

In his framework, Linos considered also programs developed with more than one programming paradigm which are defined as multi-paradigmatic programming style. In a similar way, control and dataflow dependencies are called MuLti-paradigmatic Program Dependencies (MLPD) and they are defined between elements developed with different paradigms. For example, the tuple MLPD = $< Class :: Method, Function, calls >$ indicates that class-methods calls user defined functions. An instance of this dependency is $< Shape :: draw, Printlabel, calls >$ showing that the draw method of the class shape can calls a user-defined function called PrintLabel.

Panos defined a general representation of dependencies that are suitable to each type of language (functional, Object-Oriented, logic) and also to MuLti-paradigmatic Program. So we are interested by this model because it is simple and suitable for all the paradigms and we want to reuse it by representing the dependencies as a tuple. Panos team developed the Polycare tool to implement his approach. Polycare is a tool that automates the extraction and visualization of multi-language dependencies. It allows the visualization of dependencies with several types of graphic abstraction such as the traditional hierarchical display of control flow (call graph) and a graphical representation called the colonnade, which consists of separate columns in which different entities of the program are displayed. The relationships between the entities are represented via the connecting lines between the corresponding columns. Polycare handles the interconnections between languages such as C, C++, Lisp and Prolog. However, he considered the languages in isolation in an integrated environment, in our work we want to take

into account data on interconnections that are overlapped between the two languages.

Recently, Linos presented a paper [20] to support the process of dependency comprehension and management in a multi-language systems. The research focus is specifically MLPD (MuLti-paradigmatic Program Dependencies) that appears in the interaction between the languages C, C++, and Java. In this context, he developed the MT (Multi-Language Tool) tool. MT facilitates the process of detection, storing and managing MLPDs found in programs written with a combination of C, C++ and Java. MT's GUI is based on a simple display format that uses circles, where each circle is associated with a programming language. The size of the circle corresponds to the number of lines of code. The model is animated with a gravity animation: circles attract and overlap in function to their dependencies, and also provides a zoom function. It also allows access to source code via View button-Source-Code. This tool is implemented using a lexical analysis based on the keywords of the call functions of another language other than the host language (caller). Moise [**?**] suggested that the analysis of such heterogeneous systems should rely on accurate tools like parsers as lexical analyzer may not produce accurate enough information. We think that the GUI of this tool is just suitable to have a global idea about the program and facilitate the comprehension but it is not suitable to express change impact propagation. We agree with Moise point of view that the lexical analysis is not precise and the syntactic one is better and we will investigate if we can combine the two method. For example, we can use lexical analysis for simple code patterns and for sophisticated code patterns we use syntactic analysis.

Finally, Linos et al [21] presented a tool to detect, recover, and display metrics of multi-language programs at intermediate code level. More precisely, the tool supports code written using Microsoft .Net Visual Studio. The idea is that the complexity of analysis at intermediate level is lower than if each language must be handled separately. Indeed, in a such as approach there is no need of specific parsers for each programming language. We are interested by this work because it treats particular type of multi-language program that have an intermediate language. Recently, Moise et al. proposed an approach that extracts facts of languages in a program, based on syntactic analysis of the source code [22–24]. Facts are then grouped into a common fact schema that is exploited to extract the inter-language dependencies. The approach was implemented in a tool named Clare, a plugin for Eclipse, which also includes a visualization feature. Clare supports Java, C/C++, and Perl. Moise focused more on comprehension and visualization of multi-language program. Although, we are interested on change impact analysis but the dependencies extraction is needed, so his work can helps us in this step. He used syntactic analysis that is precise method, and in our work we want to combine

this method with string based recognition.

The rest of work will be cited because of the originality of the studied system. A similar approach to Moise approach was presented by Kullbach and al. [1]. The key idea is to translate the source code of different languages into a general structure. The work of Kullbach and al. work was applied to the software of the company Aachener und Informatik Müchen System. This system is composed of several source files coded with various languages, plus a database definition and Job Control Languages (JCL): a scripting language used on IBM mainframe operating systems to instruct the system on how to run a batch job or start a subsystem. The system has the particularity of the connections between the source files that are made via JCL. So to identify the dependencies, the authors need to identify JCL files specifying interconnections. The authors also highlighted the problem of database migration that imposes to modify all files that are linked to the database. In this work, the EER/GRAL approach to graph based conceptual modelling is used. EER/GRAL is based on TGraphs (very general class of graphs). The EER is extended entity relationship dialect and the GRAL is constraint language. We cited the work of Kullbach because he handled a different case of multi-language system where the interconnections between different files and modules by the JCL, but we are not interested in this type of interconnection.

Distributed system permits heterogeneity of the system used in each component. The components can be written in different languages (Java, Cobol, C++, etc.). We will as an example of a work performed on distributed the one of Deruelle et al. [25]. They investigated the analysis of distributed multi-language software applications. They proposed a formal model called *Source Code Structural Model* (SCSM) based on UML. SCSM models components and their interconnections. JavaCC was used to develop a parser what generates an XML (eXtensible Markup Language) representation of the SCSM diagram. This diagram is used to implement two modules one supporting the management of changes and a profiling module. The change management module propagates the effect of changes by revisiting and modifying relevant nodes of the SCSM diagram. The profiling module measures the contribution of a component to the overall software performance.

Hassan [26] was interested in Web application architectures that are generally multi-language. His goal was to develop a tool that assists developers in understanding the structure of their web application. For that he used a three-step approach:

- Extracting facts from source code of a program using a set of extractors: he has developed five types of extractors: HTML extractor, script server extractor and Access DB extractor. These extractors are managed by a script that determines the type of a file and then invoke the corresponding extractor.
- Abstraction and fusion of multi-language facts.
- Generating the diagram of the architecture.

This work is focused on extracting the architecture of a multi-language Web application not the dependencies between different components. We will use a similar approach to determine the type of the file but we want to combine with this approach island grammar technique to distinguish different language in the same file.

3.2 Licenses analysis

The majority of the effort in research target technical problem of the software development and re-engineering and a little attention is directed to the legal complexity [10]. When the developers combine several components with different licenses to create program, the possibility of having licenses mismatch increases. In addition, software engineers have limited knowledge of legal issues.

We will present the earliest work on licenses analysis that also influenced our project.

German et al. [14] models a license as a set of grants, each of which has a set of conjoined conditions necessary for the grant to be given. The compatibility of licenses is analysed by examining pairs of licenses. They considered five types of interconnection (linking, fork, subclass, IPC, plugin) and they developed a model that describes the interconnection of the components. They identified twelve patterns for avoiding licenses mismatches, found in a large group of OSS projects. They used their models to document integration patterns that are commonly used to solve the license mismatch problem in practice. In our work, we use license model of German in [14] and we consider the same interconnections types to analyse the compatibility of licenses, because it simplifies the process of analysis and distinguishing between derivative and collective work.

German et al. [10] proposed a method to help the understanding licensing issues that can arise from changing, combining, and re-distributing packages in open distribution. They carried a large empirical study aimed at analyzing licensing issues in the entire Linux-based Fedora-12 operating system. They considered constraints imposed by open source

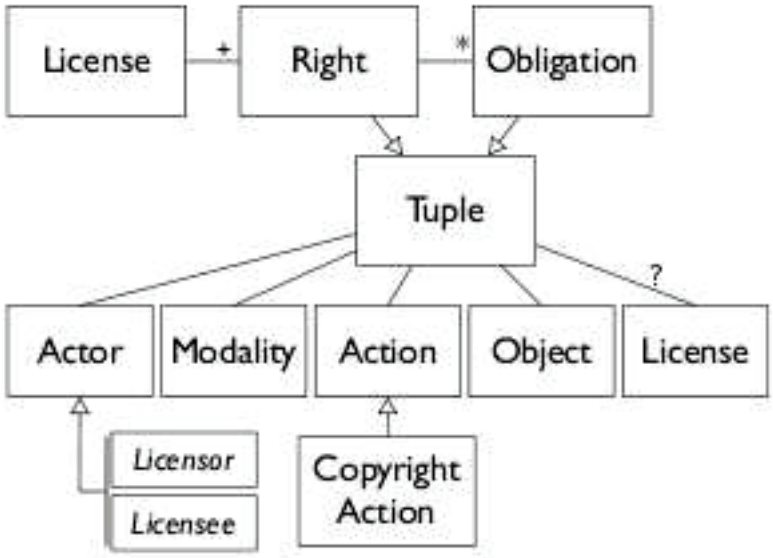


FIGURE 3.1: The metamodel for licenses

licenses and rely on it to mine inconsistencies between licenses declared in the packages and source code licenses and incompatibilities due to the dependencies between packages and libraries having different licenses. They identified the license and dependencies of all files using RPM package descriptions but the identification of dependencies types is done manually. We want to extend this work by decreasing the effort of the manual identification of interconnection type.

Alspuagh et al. [27] performed parametrization analysis based on semantic parametrization of nine OSS licenses. From this analysis and patterns identified and the models established by German, they derived the meta-model for licenses shown in figure 3.1. Their license model extends German's to address semantic connections between obligations and rights. They developed a tool that supports intellectual property requirements management. The main advantage of this tool is the ability to model software systems at different architecture levels and to analyze license interactions across the different architecture level.

Tuunanen et al. [28] proposed a comprehensive approach for supporting software license analysis. Their approach is implemented in tool called ASLA that identifies licenses from source code, uses compiling information by using GCC and also ar (an archive tool) and Id (a linker) to determines if two components are connected together to find violations of licenses constraints. The license identification is achieved by using license templates given as regular expressions. Simple open source license such as BSD and MIT are often included at the beginning of each source code file. But this exact matching does not work very well with real source code file because of many reasons, e.g., comments and

various kinds of white space characters prevent exact matching and many programmers modify the predefined text and there are different published versions of licenses.

Both Tuunanen [28] and Alspaugh [27] has evaluated their approach on small applications composed of few components. In the other hand, German and al. [10] deal with large Linux distributions containing more than 10,000 binary applications and hundreds of thousands of source code files (Fedora-12). We will also extend the work of German and al. [10] like we mentioned before to automatise the identification of interconnection types and then to analyse change impact.

Chapter 4

Approach

Many previous work studied the impact of changes on technical characteristics, in our work we want to support program evolution via change impact analysis of heterogeneous systems and we want also to know the impact of change on the legality of the system. Also, we would like to recommend possible architectures that can be obtained by combining components written in C/Java/Perl and having different licenses that can be Modified BSD, Apache, GPL.

In this chapter, we describe the steps to follow to answer our research questions.

4.1 RQ1: What is the change impact analysis on multi-language program?

Our first research question: How we can analyse the impact of modification of an entity (method, file...) in multi-language system? We follow two directions to answer: the extraction of dependencies and the implementation engine that will give us the change impacts. In the next two sub-section, we present our approach to resolve these question see Figure 4.1. The input of change impact analysis is the source code of the software that we want to analyse and the change specification and the output will be a graph that represent the impact. This graph can be the entities that are dependent on or use the element(method, variables...) changed and must be updated if we apply the change.

4.1.1 Dependencies extraction

1. First, we must identify the existing language of the source code. This task can be done with a file navigator. The file navigator will reach all files in the software and

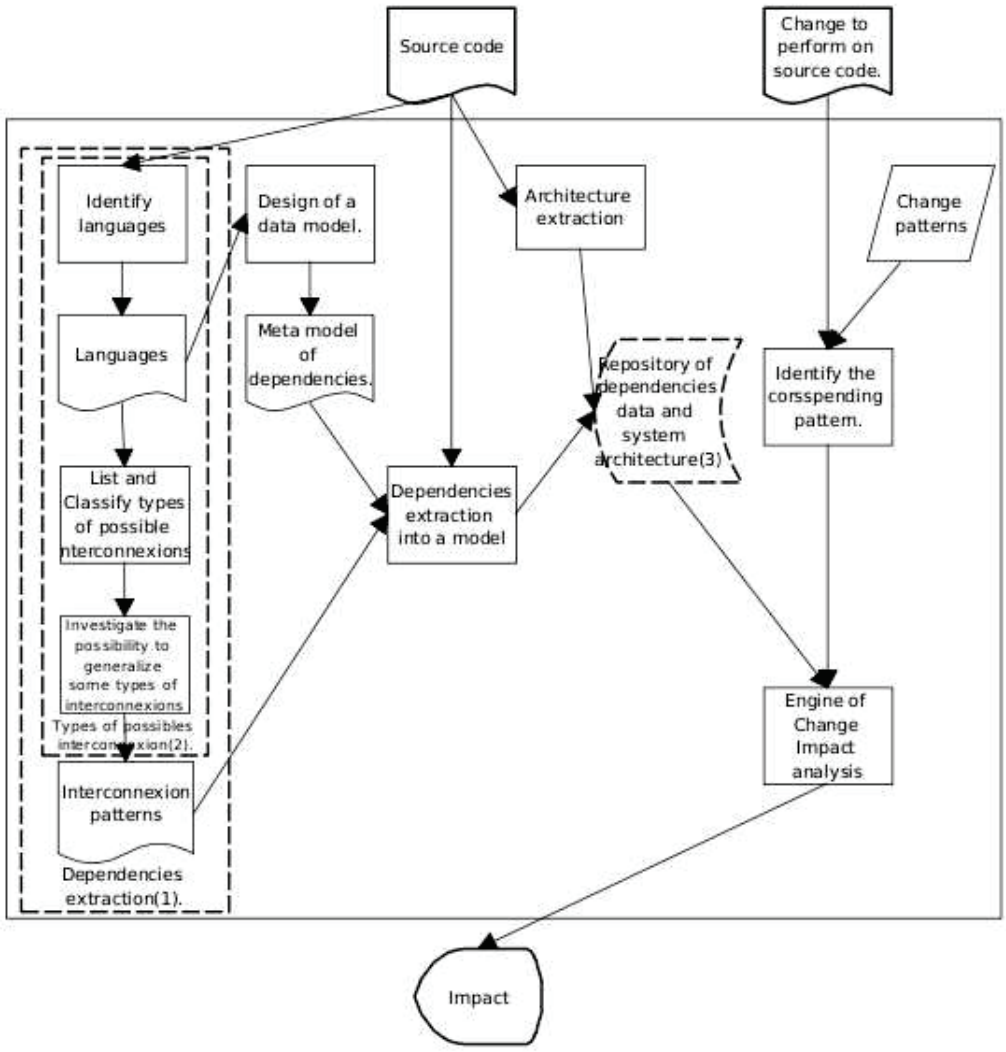


FIGURE 4.1: RQ1: Change impact analysis

try to identify the language of each file using the extension of the file if it can't identify the languages then we will attribute unknown status. The input of the file navigator will be a list of languages and their associated file extensions and the output will be the list of all files in the system, their languages and associated extractor. We denote LG the set of existing languages in the source code. To take on to account files with mixed language, we think that we will use analyser based on island grammar [29][26].

2. Then, we list and classify manually the types of interconnection between different languages. In this step, we want to classify the interconnection types like linking, exec/fork,... For each $lg_i \in LG$ and $lg_j \in LG$, we will do a literature review to find how the language lg_i can call the language lg_j.

3. Specify a common data model or linked data models that supports the concepts of multiple programming languages and models dependencies.

4. Having the possible interconnection between different language, we will investigate the possibility of generalizing the method of extraction of certain types of dependencies for pairs of different programming languages, for example, it is possible that we found call executable file performed by similar instructions in several languages so we can extract it by the same analyser. For this step, we will reuse the result of the previous step: the possible type of interconnection, we will analyse the similarity between these types of interconnection, and classify the similar one together in order to try in the next step to assign to them the same pattern.

5. We want to abstract the code that define formally the link between language. The abstraction will be based on code patterns. These patterns distinguish clearly each type of interconnection. Each pattern has typical properties and elements which indicate it. These patterns will be used to recognize dependency in the source code. The design of a meta model is needed to support the representation of dependencies between the set of languages LG identified at the first step. The meta-model must support the representation of dependencies of all the types of interconnections that can be used in LG. We suggest the meta-model PADL [30]. PADL (Pattern and Abstract-level Description Language) is a meta-model that describes the models of program source code using AOL [31], C++ and Java (including AspectJ) analyzers. This model supports programs written in AOL, C++, Java, we want to study the possibility of generating a PADL meta-model for a multi-languages program. This model will be used also to find the propagation of the impact. The propagation of impact represent the set entities that are affected directly or indirectly by the change.

6. We need also to extract the architecture of the system. To perform it, we will choose a convenient tool that use an efficient model. The meta-model PADL can be used also to represent the system architecture.

7. Finally, having meta-model that supports dependencies and and the interconnection patterns, we have to search and choose an existing String based recognition tool to recognize the code corresponding to the interconnection. The input of these tool is the pattern of interconnection formatted in the format requested by the tool and the source code. Perhaps, this tool can not be sufficient and precise to recognize all the type of interconnection, as alternative we will use a syntactic analyser for the corresponding programming language.

4.1.2 Implementation of Change impact analysis tool

The goal of this step is to develop a tool which will give us the impact of a change. The input of this tool will be the change pattern and the repository of dependencies data extracted at the first step and system architectures extracted in the previous step.
We want to specify the change that can be done on a program. So we will have a list of predefined class of change types and their specifications. And we want to associate to each change type an impact pattern to predict the types of consequences formally. For example: when we rename a method: the pattern is renaming method, then we have to search all sites which call the method and replace the old name of the method by the new one.

4.2 RQ2: What are the possible architectures of a heterogeneously licensed system?

Heterogeneous software can combine components with different licenses. Their compatibility depends on their licenses and the interconnections types used as a glue. We would like to present possible architectures that can be obtained by combining components written in C/Java/Perl and having different licenses that can be Modified BSD, Apache, GPL.

In our research we plan to apply the following steps, see Figure 4.2:

1. Identification of the different modes of interconnection between programs written in C (procedural), Java (OO), Perl (script). This step is expanded in the Figure 4.1 (see box 2) if the languages concerned exists in the source code studied at the first question.

2. Identification of grants for each license: Modified BSD, Apache, GPL. Establishing inter-license compatibility rules, i.e, under which conditions can the two licenses be used together. Normally we must read the terms of different licenses and try to interpret and extract their constraints using the grants and the conditions. The licenses interpretation is not evident task. Specially in the case of bare license (simple license without doing a contract between the licensor and the licensee), there is no ready framework for license language interpretation like when the GPL does not even demand acceptance of terms of the license, can a licensor assume that licensees have agreed to all of those terms? The courts around the world, don't agree on what constitutes a derivative work of software. Even if we suppose that the licensee accepted the terms of the license, what about terms in license that

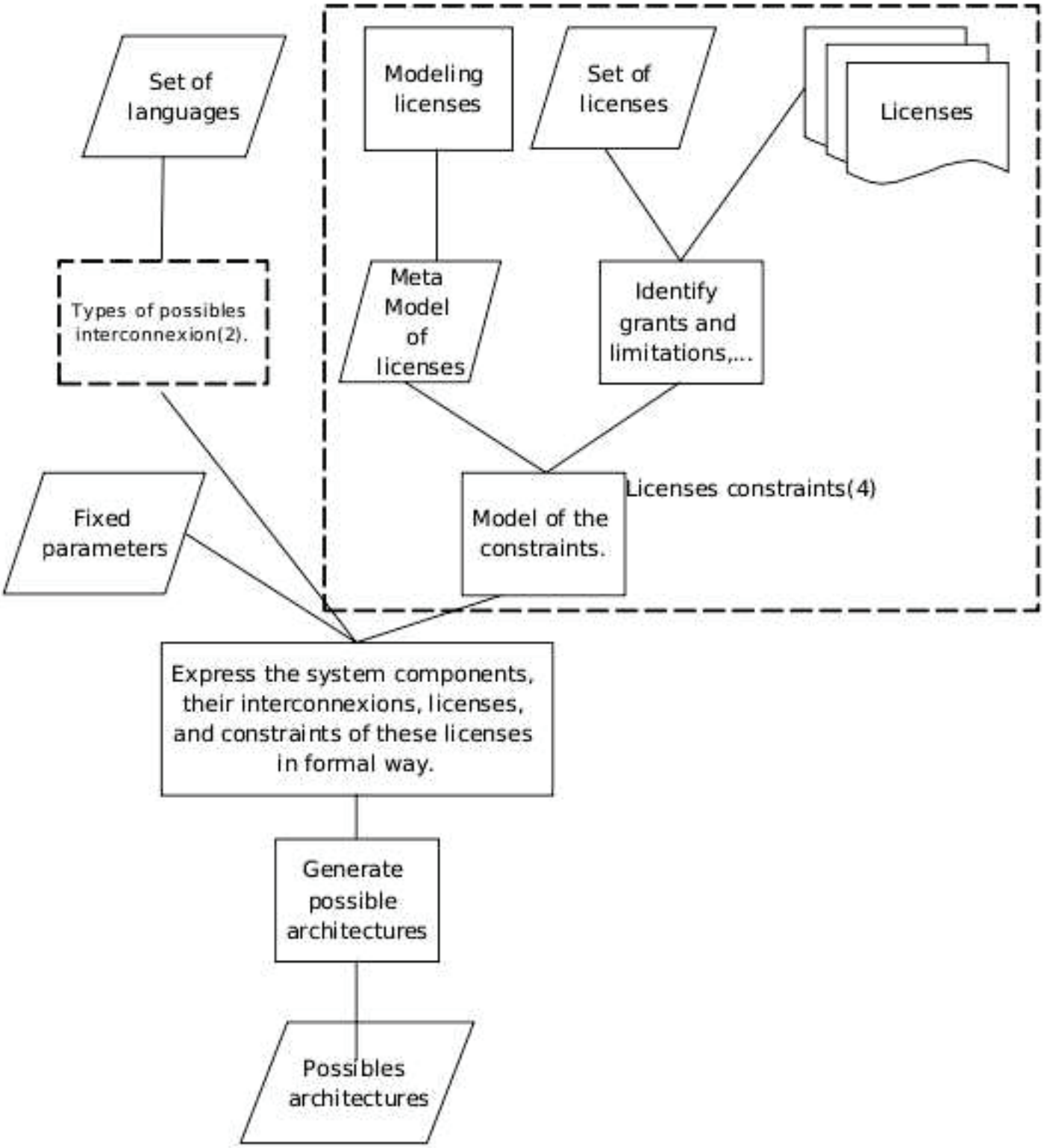


FIGURE 4.2: RQ2: The possible architectures of a heterogeneously licensed system

are inconsistent with the definitions of art in copyright law, such as derivative work [4]? So, we will not use directly the licenses terms but we will do a literature review of document produced by the experts in this domain to retrieve the constraints of each license.
After, retrieving information about inter-licenses grants and limitations, we want to express them formally by using a meta-model of licenses description.

3. A system is a set of components written in various languages and with various licenses. The variables of the system are languages, licenses, interconnections, and their types. We want to choose and fix the values of these variables and the rest of unspecified variables their values will be suggested by our tool us output. We notice that this problem can be expressed by IF THEN productions rules. So, a system expert could be a good technique to develop a tool which gets as input a Fact Base (FB) where all information about the system are stored and the Rule Base (RB) that based on possible interconnections between the languages (step 1 of RQ2) and the constraints and licenses compatibility (step 2 of RQ2). the rules can be expressed as a propositional calculus or logic (also called sentential calculus) that is a formal system in which formulas of a formal language may be interpreted as representing propositions (zero-order logic) or predicate logic in which formulas contain variables which can be quantified (first-order logic). In our case the zero-order logic is sufficient because we do not need a quantifiers for variables, so the propositional calculus will be used to express rules. Then the Inference Engine(IE) makes inferences by deciding which rules are satisfied by facts, prioritizing the satisfied rules, and executing the rule with the highest priority. In our case, the rules determines the interconnection between components, their languages, the licenses and their constraints.

4.3 RQ3: What is the impact of source code change on license compatibility?

Going back to our goal of investigating the impact of a component modification, our other goal is to make the developer aware that certain modification may violate licensing constraints. When we perform a modification on some software components, we argue that such modification can affect the legality of the system by causing licenses violations. For example, when a component is updated to a new version, the new version can have a different license than the old one, this license can be incompatible with the license of the system. If we modify how a component connect to another component can introduce a violations, for example we have a system S licensed under Modified BSD that calls a

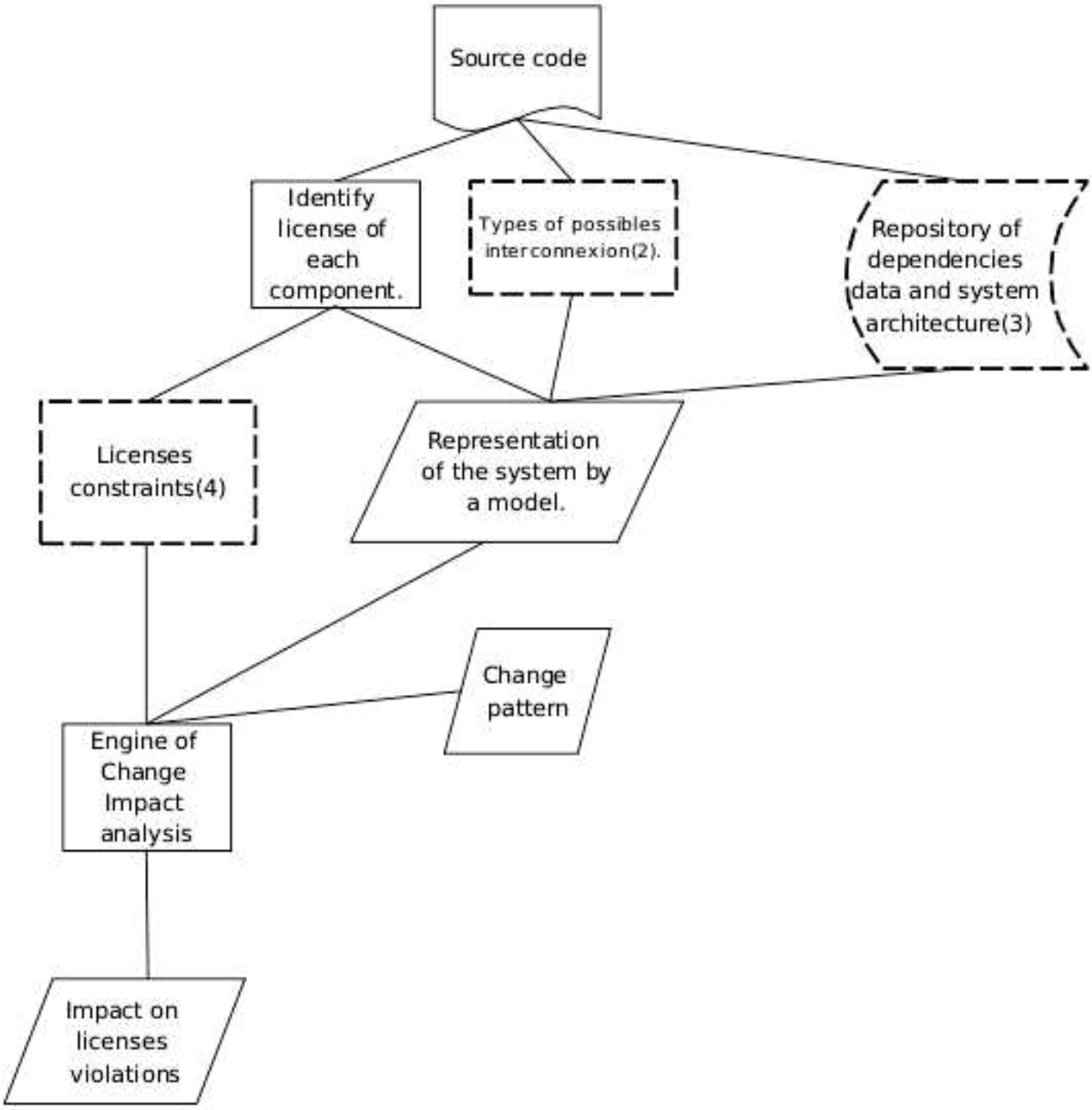


FIGURE 4.3: RQ3: Change impact on the license compatibility

component C^* licensed under GPL via black-box linking and we modified the system by changing this interconnection to white box linking this is a violation of the GPL license because the system in this case will be a derivative work of C^* and must be licensed under the same version of GPL of C^*.

To answer this question (RQ3), we will combine the two previous work by using results from RQ1 about components and the interconnections between them and from RQ2 about the licenses of the system components. We must extend the model established at RQ1 by adding the support of licenses representation. Then, the licenses of components in the system must be extracted in to the repository. We will reuse the identification of grants for each license in RQ2. Indeed, having the structure of the actual system (licenses, interconnection, language) and the component (license, language) that we would like to change or add will allow us to reason about the possible violations of licensing. If there is a violation we would like to suggest the possible legal architectures to eliminate this violation, this can performed by using RQ2.

Chapter 5

On going work

We presented our goal and the related research questions. We showed how we will resolve these research questions. In this chapter, we present on going work. We begin our work around the problematic of the constraints introduced by licenses heterogeneity in the system. In the first section, we will approach RQ2: What are the possible architectures of a heterogeneously licensed system? In the second section, we will present the work of Daniel et al. [10] that proposes a method to understand licensing compatibility issues in software packages, and we will show how we want to extend this work by decreasing the manual effort to identify interconnection types. This work will be a preliminary work to answer the research RQ3 about the change impact on license compatibility.

5.1 Possible architectures of a heterogeneously licensed system

We would like to present possible architectures that can be obtained by combining components written in C/Java/Perl and having different licenses that can be Modified BSD, Apache, or GPL. The first step to reach this goal is investigating the interconnection types between the languages: Java, C, Perl. That's why, we did a literature review to find almost types of interconnection that will presented in the next section and in the appendix 7. Then, we identified the grants for each license: Modified BSD, Apache, GPL presented in the introduction 1. And we established the inter-License compatibility rules,i.e, under which conditions can the two licenses be used together. As we proposed in our approach we will use system expert to develop a tool that gives us the possibles architectures of a heterogeneously licensed system. So, we will formalize the interconnection between components, the license of component and, the language of each component that will permit to represent facts base, and we will formalize inter-license compatibility

and possible interconnections between different languages to represent rules base. So this formalization will provide the input of system expert.
In the next section, we will present almost of interconnections between Perl, Java, and C. Suppose that a system S call a component C, we considered four types of interconnection proposed by German and al. in [14]:

- Linking: Calling functions or methods in C using dynamic or static linking.
- Fork: Stand alone execution via fork or exec system calls. C is executed in a separate space from S. The communication between the rest of S and C might be done via pipes, sockets or files.
- IPC: C is built as server or server. Other parts of S use C via well-defined process intercommunication protocol, such as CORBA and COM.
- Plugins: S extends the functionality of C using C's plugin-architectures.

In the second step, we will present the formalization the interconnection between components, the license of component, the language of each component, the inter-license compatibility, and possible interconnections.

5.1.1 Interaction between Java and C

5.1.1.1 Calling C program From Java program

Linking We must follow these steps to call a C program from Java [24]:

- Create a Java file that declares a class with one or many native method
- Compile the Java file to create a .class file.
- Use the generator javah with the option *-jni* to create a header file to use in the C program.
- Create a C file which implements the native methods.
- Compile the C file to create the library of dynamic link to export the native method.
- Execute the Java program.

Example of a Java file that declares a native method:

```
  Public class test {
     static  {s.loadLibrary("Test");}
     public int iValue;
     public Test() {}
     public double compute (Vector v, float f){}
     public native void print(String msg);{}
     public static void main(String[] args) {
      Test t = new Test();
        t.print("Hello from C!");
     }
 }
```

Test class contains the native method *print*. This class load dynamically the library that provides the implementation of the method *print* in C. When, we execute the generator *javah* with the option *-jni* to the file Test(.class), we obtain the file test.h that contains the declaration of the function corresponding to the native method *print*:

```
void JNICALL Java_Test_print(JNIEnv *, jobject, jstring);
```

The name of the method which correspond to the native method consist of : 'Java', '_', and the name of the native in Java (for example, print). Also, its signature consists of three arguments : the first is requested by the JNI to access to its functions. The third is *jstring* which corresponds to Java argument of the native method *print* that has String type.

Fork If we want to call external programs (executable program) in a Java application, we can use system call by creating Runtime Object and attaching it to a system process [32].

Example:

```
import java.io.*;
public class Main {
       public static void main(String args[]) {
            try {
                Runtime rt = Runtime.getRuntime();
                Process pr = rt.exec("c:\\helloworld.exe");
                BufferedReader input = new BufferedReader
```

```
                            (new InputStreamReader(pr.getInputStream()));
                String line=null;
                while((line=input.readLine()) != null) {
                    System.out.println(line);
                }
                int exitVal = pr.waitFor();
                System.out.println("Exited with error code "+exitVal);
            } catch(Exception e) {
                System.out.println(e.toString());
                e.printStackTrace();
            }
        }
}
```

Method *waitFor()* will make the current thread to wait until the external program finish and return the exit value to the waited thread.

IPC C can use Signals, Pipes, Messaging Queue, Semaphores, Shared memory, Socket. Java can use Signals, Semaphore, Pipes, Shared Memory, Domain Socket, RPC (remoting), Socket (UDP or TCP). So Java Program can call C program using Signals, Semaphores,Pipes, shared memory, Socket.

5.1.1.2 Call Java from C program

Linking A C function can create, update and access to Java objects. There are two methods to access a Java program from a C program. The first method is a Java method is implemented in C et and the C method call back the Java object. The second method is to embed the Java Virtual Machine (JVM) [33]. In the two case, it is the JNI API that supports the communication [24].

We reuse the same example of Test class. We modify the C function Java_Test_print which implements the native method *print* to access to *iValue* and *compute*, the members of Java class.
The corresponding code is :

```
JNIEXPORT void JNICALL Java_Test_print(JNIEnv *env, jobject obj,
```

```
jstring s)
{
jclass cls = (*env)->GetObjectClass(env, obj);
jmethodID mid = (*env)->GetMethodID(env, cls, "compute",
                                    "(Ljava/util/Vector;F)D");
    ....
(*env)->CallDoubleMethod(env, obj, mid, v, 1.0f);
    jfieldID fid = (*env)->(env, cls, "iValue", "I");
int i = (*env)->GetInField(env, obj, fid);
  ...
}
```

The Java class is retrieved using the JNI function GetObjectClass. We will use GetMethodID with the name of Java class and its signature. To find the wanted method, JNI search in the symbol table using the name of the method. The method is invocated using the JNI method Call<T>Method , where T denotes the type returned by the method. The value of the field iValue is accessible via the method GetFieldID and respectively the JNI function GetIntField [24].

Fork To execute a Java program or to call an executable from C, it is possible to use the "system" function:

CPP / C++ / C Code:

```
 int result;
//Execute command.
result = system(Java_program);
```

"your_command" is the name of the command / filename that we want to execute.

IPC C can use Signals, Pipes, Messaging Queue, Semaphores, Shared memory, Socket. And, Java can use Signals, Semaphore, Pipes, Shared Memory, Domain Socket, RPC (remoting), and Socket (UDP or TCP). So, C program can call Java program using Signals, Semaphores, Pipes, Shared Memory, Socket.

Plugins It is possible to write plugins with Java for C program. For example, collectd software [34], it is a daemon that collects system performance statistics periodically and

provides mechanisms to store the values in a variety of ways. It is written in C language and It has Java plugin to it. The Java plugin embeds a Java virtual machine (JVM) into collectd and exposes the application programming interface (API) to Java programs.

5.1.2 Open Source Licenses

We define a component-based software application(S) as a work composed of one or more software components (C_i) functioning together. Each component has its own copyright owner (who can be the end user or the integrator putting S together) and its own license ($L(C_i)$). Similarly, S can have its own license L(S) [14]. A software system S can be modeled as dependency graph with each component (C_i) as a node and the edge presents interconnection between components. The interconnection can be through as we mentioned before : linking, fork, subclass, IPC, and plugin, for the example see the figure 5.1.

To determine if there a violation, who connect to who is not important, for example C_3 that licensed under BSD-3 calls dynamically C_2 that licensed under GPLv2, normally it is not permitted but when all this component belong to final system S we don't care of the inter-relation between components as separate system. But what matters is the type of interconnections indicated in the outgoing arrow and the license of destination components. In our example, we are interested to:

$\longrightarrow$ exec call to BSD-3

$\longrightarrow$ Dynamic linking to GPLv2

$\longrightarrow$ exec call to BSD-4

We are not also interested to the license of C_4 that is GPLv2+ because there is no incoming link to it. So, the final license of the system must be GPLv2 or any later because there a dynamic link to GPLv2 licensed component.

5.1.2.1 Modeling open source licenses

An open source license provides its licensee with grant to one or more of the exclusive rights owned by the copyright owner of the component. Each grant is granted provided

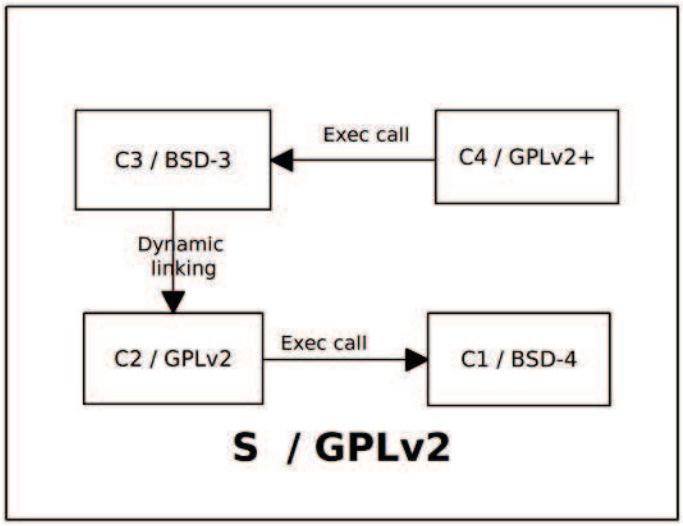


FIGURE 5.1: Example of heterogeneously licensed system

a set of conditions are satisfied, all these conditions must be satisfied [5].
To model and open source licenses, we will reuse the model in the work of Daniel M.German and al. [14], a license is set of grants. *"The conditions for each grant to right r (Gr) can be represented as a set of m conjuncts should be satisfied for the licensor to receive such grant:* $Gr(L) = p_1 \wedge ... \wedge p_m$. *If one of the conjunct* $p_1 ... p_m$ *are not respected so the grant of the licenses is not given. So, we have to identify and interpret grant that exist in licenses and there conjuncts."* [14]

5.1.2.2 Modeling licenses compatibility

To formalize also the compatibility of licenses also using the work of German and al. [14]: *"Each of the rights needed for S will require a grant to one or more specific rights from each of C1 ...Cn . Each grant imposes a set of conditions, which are modeled as conjuncts. If the union of all these conjuncts is not satisfiable, then I cannot acquire the desired rights and might not be able to create S or license it to anybody. We say that, for a given grant g, L(C) is not compatible with L(S) if at least one of the conjuncts of g from L(C) is not satisfiable under L(S). We denote this relationship as* $_g$ *(compatible) and* $_g$ *(not compatible): if under grant g, L(A) is compatible with L(B) then A* $_g$ *B. By extension the use of C in S is compatible under use u (denoted C* $_u$ *S) iff the use u is permitted by a grant g of L(C), and L(C)* $_g$ *L(S)."*

5.1.3 Software Architecture

Actually, we will use a simple model [35] to represent the system architecture.

5.1.3.1 Definition

They are many definitions for software architectures, One of the earliest is proposed by Perry and Wolf(1992): "a set of architectural elements that have a particular form" they propose three types of elements: processing, data and connecting. Connecting elements distinguish one architecture from another. The form of the architecture is given by enumerating the properties and relationships of the different elements[35]. And one the newest definition is offered by Bass, Clements and Kazman(1998) offer the following definition: *"Software architecture is the structure of the structures of the system, which comprise software components, the externally visible properties of these components, and the relationships among them".*

5.1.3.2 Software architecture elements

A software architecture is comprised of elements [35]:

- Components:
  Components are the basic building blocks and the active, computational entities in the system. Components accomplish tasks through internal computation and external communication.
- Connectors:
  Connectors define the interaction between components an describe the rules that govern those interaction.
- Interfaces:
  A components defines an interface through which a connector links one components with another.
- Configuration:
  A configuration (called topology also) is a connected graph of components and connectors that describes architectural structure

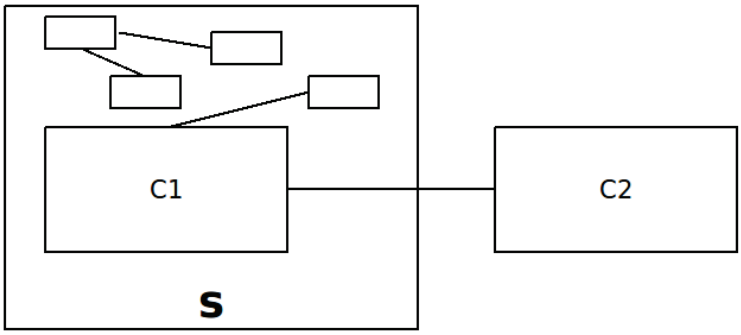


FIGURE 5.2: Architecture model

5.1.4 Possible Architectures

In this section, we suppose that we have a system S: a component-based software. S is composed by one or many components C_i, $i \geq 1$. And C^* is the program that we want to add (to use in) the system S. C^* is connected to S necessarily with at least one component of S. We must analyse the connections of S and C^* one by one. We suppose that C^* is connected to S just once because in the case of many interconnections, the handle of all interconnections is equivalent to the handle of the sum of interconnections (one by one), see figure 5.2.

5.1.4.1 Example of possible architectures for program written with Java and C

We want to show how we can found possible architectures manually for a special case of program written with Java and C and we will consider these licenses: Modified BSD, GPL, Apache. First, we will present some notation and we will describe the system characteristics. Second, we will show how intuitively we can find the possible architectures. From this manual method we will deduce will present the system that respect the constraints. In the next section, we will explain how we can automate this process using system expert. Notation:

PL(C) denote the programming language of the component C and L(C) is the license of the component C.
We suppose that we have a system S and and the component that has to be added to S is C^* and $PL(C^*) = C$ $L(C^*) = GPL$. It will be connected to the component C_1 and $PL(C_1) = Java$

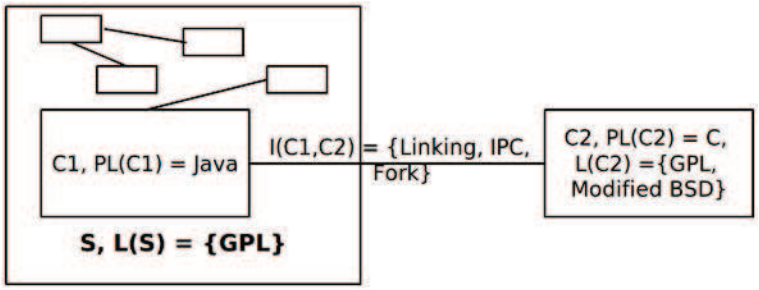


FIGURE 5.3: $PL(C^*) = C$, $PL(C_1) = Java$, $L(C^*) = GPL$

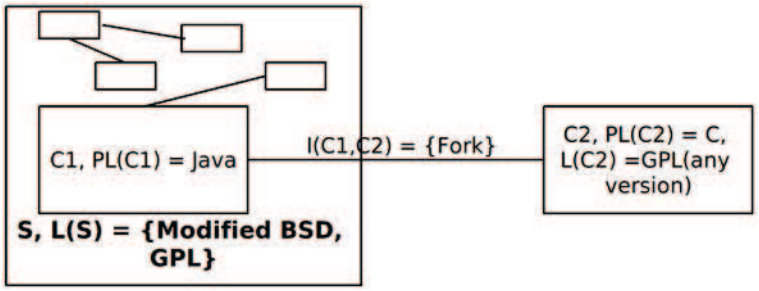


FIGURE 5.4: $PL(C^*) = C$, $PL(C_1) = Java$, $L(C^*) = GPL$ and $I(C_1, C^*) = fork$

Knowing that the Java can be connected to a C program with the following type of interconnection : linking, fork, IPC, we want to know what is possible license of S $L(S) =?$ and the interconnection type between C_1 and C^*. The answer of these question will be a set of a couple $(L(S), I(C_1, C^*))$

If the interconnection type is linking or IPC, so S is considered as derivative work of C^*, then necessarily L(S) must have the same license as $L(C^*)$ with the same version, so L(S) = GPL and the solution is $(L(S) = GPL, I(C_1, C^*) = linking, IPC)$ see figure 5.3. In this case, we can also another sub case that can be added to the same figure, to have a compact presentation, when we have $L(S) = GPL$, we know that is possible to connect to it any component which has a GPL compatible license, in our case is GPL itself and Modified BSD.

Else if the interconnection type is fork so S can be licensed under any license compatible with GPL in our case is GPL itself and Modified BSD see figure 5.4. There is sub case, when the $L(S) = GPL$ and the type of interconnection is linking or IPC or Fork, we can have $L(C^*) = ApacheV.2$ but in he condition that L(S) is exactly the GPLv2.0 because only Apache V.2 and GPLv3 are compatible see figure 5.5.

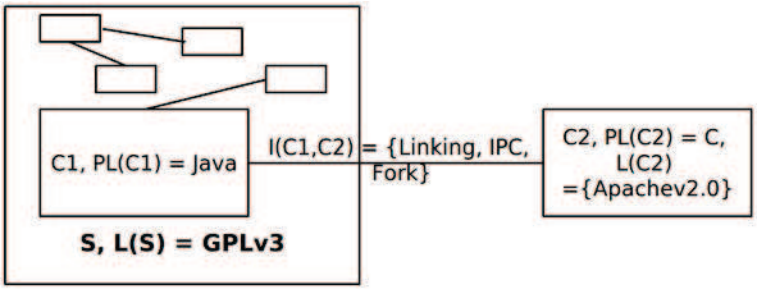


FIGURE 5.5: $PL(C^*) = C$, $PL(C_1) = Java$, $L(S) = GPLv3$

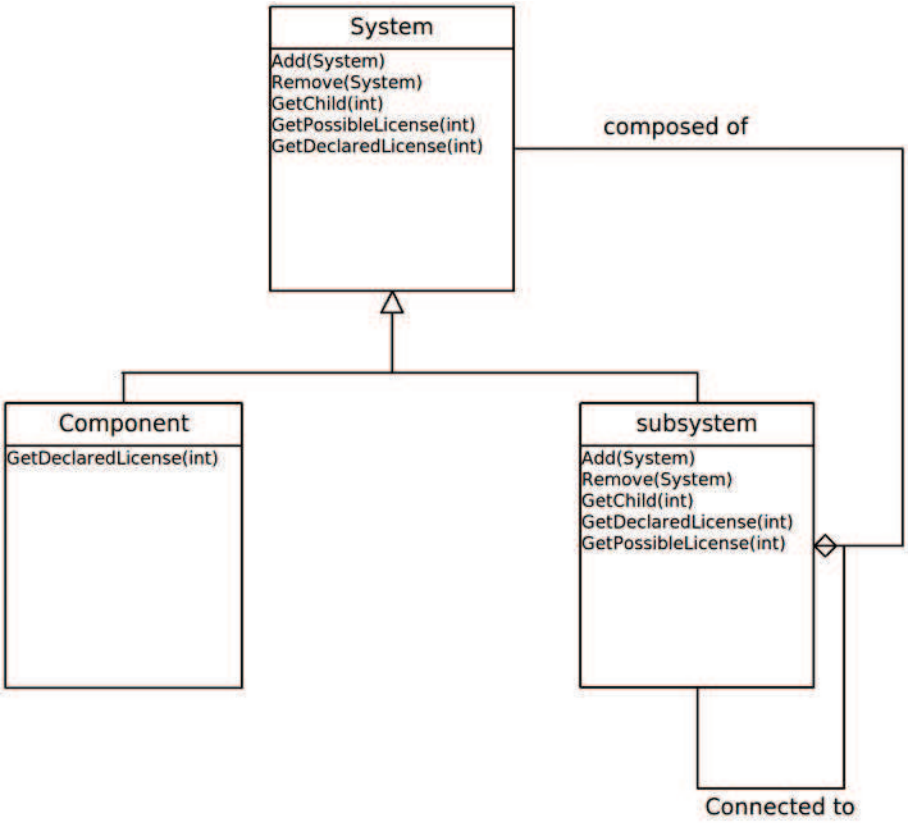


FIGURE 5.6: MetaModel of the system

5.1.4.2 Formalisation and definition

In the figure 5.1.4.2, we present the meta-model of the system. The system is composed of several subsystem and a subsystem C_1 can be connected to another subsystem C_2 to have another subsystem. And the atomic subsystem is component.

The whole system S is denoted:

$S = C_i \cup S_j$
such as C_k such as $k \in \{1..n\}$ and $n > 1$ denote a component
$S_j = \cup C_k$ denote a subsystem.

SS : the set of the components

$CON = \{linking, fork \setminus exec, IPC, Plugin\}$: the set of possible interconnections

License: $\cup C \longrightarrow L$
$S_j \longrightarrow L(S_j)$

Interconnection: $SSXSS \longrightarrow CON$
$S_i, S_j \longrightarrow I(S_i, S_j)$

We define the license of components and in general sub system like that:

License: $SS \longrightarrow \cup C$
$S_i \longrightarrow L(S_i)$

Compatibility of licenses:
Compatibility : $LxL \longrightarrow \{0, 1\}$
$L1, L2 \longrightarrow CP(L1, L2) = 1$ if L1 and L2 are compatible else 0

Example of question:
What is the possible interconnection type if you want to add C^* to S knowing that C^* has a license L^* and written with Language LG^* and it will be connected to C1 that is written with language LG1 and the system S has a license L. *Answer:*
We suppose that we don't want change the licenses of S and C1.

If L^* and L are not compatible so $I(C^*, S) = \emptyset$

If L^* and L are compatible so $I(C^*, S) =?$

We must verify The conditions of the license L^* for the grant of doing a derivative or collective work $Gr(L) = p_1 \wedge ... \wedge p_m$, These conditions must be satisfied, we can deduce the good value of $I(C^*, S)$.

This example of question are made manually to show the intuition behind this analysis and how it lead us to think that expert system are suitable to develop a tool to automate this process: The use of propositional calculus to find the answer, the use of facts to describe the system.

5.1.4.3 System expert

The next step is to develop a tool in order to automate the process of listing the possible architectures. As we proposed in our approach, we suggest to use a system expert. The formalisation done in the previous step will be used to define the fact base and we have to add the rules that permit to inference engine to produce new facts until having the answer.

5.2 Preliminary work for change impact analysis: Extraction of dependencies and types of interconnection

German and al. [10] propose a method to understand licensing compatibility issues in software packages, and reports an empirical study aimed at auditing licensing issues in binary packages of the Fedora-12 GNU \ Linux distribution. Their objective is is to understand how the licenses declared in the packages are consistent with those of the source code files. And to audit the licensing information of Fedora-12, highlighting cases of incompatibilities between dependant packages. They followed this steps to accomplish their goal:

- They identified the licenses and the dependencies of all files using package descriptions: They extracted information from package management system of a GNU/Linux distribution. In this context, For each source packages, they extracted .spec file which is parsed to extract dependencies information and declared license. Then they used Ninka license identification tool to classify their licenses [36]. Ninka uses sentences-based approach to detect the presence and identify open source licenses in the header comments of source code file.
- Then, they combine the dependency graph of a binary package with the declared licenses of its dependencies.

We notice that the identification of dependencies types is done manually. We would like to decrease this effort by adding a functionality that helps to determine a maximum of interconnection type. We begin with the simplest interconnection and we will iterate after with the rest of dependencies types. We want to adopt an iterative method each step will identify a type of interconnection and in the next iteration we apply the analysis in the rest of unidentified interconnections. We notice that identifying system call between different files is the simplest one, so we can begin by identifying this type of interconnections by using String based recognition. In the next iteration, we see that Fedora-12 contains a lot of C files, so identifying linking interconnection is can be simple if we use the GCC.

The detailed steps are:

- Identification of fork interconnection:
  We are inspired from the work of Martin Pinzger and al. [37]. In this work, the authors present an approach that uses source code structures as patterns and introduced an iterative and interactive architecture recovery approach built upon such lower-level patterns extracted from source code.

  Steps:

  - Pattern identification: Find key information(patterns) which enables the description of interconnection properties of a type of interconnection that we want to extract.
  - Use pattern definition language which facilitate regular expressions and source code structures. There are several tool for string-based source code analysis(Perl, grep,..) but they do't support structures, that's why the authors developed a new tool named ESpart to allow pattern specification in XML. In our case, we can use grep to identify system call in C language since the pattern of system call in C language is simple.

- Identification of linking interconnection:

  1. Identification of the program modules: sources files and their types. We got the Fedora-12 binary. It is composed by 1,475 source code packages. In order to identify all the files in this packages, we wrote a shell script which tours recursively the folders and decompress recursively also the archives to take

in to account all depth of compression(more than one level). But the great number of packages make the control of it difficult so we decided to begin our work with some packages. We must choose the relevant packages that we will study by doing a simple grep to see what is the packages that have many interconnections.

2. In order to identify linking dependencies of each file, we will use GCC to identify the dependencies of each file which can be compiled by GCC. We tested GCC under.cpp test files by executing this command : $gcc - MMD * .cpp$

   The option -MMD is like -MD option except mention only user header files, not system header files. And MD option is equivalent to -M -MF file, except that -E is not implied.

   - The -MF option : The driver determines file based on whether an -o option is given. If it is, the driver uses its argument but with a suffix of .d, otherwise it take the basename of the input file and applies a .d suffix.
   - And -M option output a rule suitable for make describing the dependencies of the main source file. The preprocessor outputs one make rule containing the object file name for that source file, a colon, and the names of all the included files, including those coming from -include or -imacros command line options. And -MF permit to specify a file to write the dependencies to.

   It produces information about source code dependencies by generating .d files that contains information such as:

   ```
   main.o: main.cpp setVector.h listTabou.h gammaMoins.h gammaPlus.h Teta.h
   ```

   We must execute this command to all cpp and c files in the codes sources of Fedora-12: tour all the folder in Fedora-12 and execute this command to all cpp and c files in each folder. The output of this command is a . d file for each c or cpp file.

3. After, we have to parse all this .d files to extract the dependencies that will be exploited to constitute the dependency graph of the program in terms linking.

Chapter 6

Conclusion and planning

6.1 Planning

The table 6.1 describe the planning of our project and the conference in which we want to present our results.

	W09	S09	W10	S10	F10	W11	S11	F11	W12	S12	F12	Publications
RP and courses	x	x	x	x								
RQ1				x	x			x	x			WCRE
RQ2				x	x	x						ICSE/TSE
RQ3				x	x	x	x	x	x			ICSM/JSME
Writing										x	x	
Defense											x	

Table 6.1: Planning of the project

Key: F: Fall, W: Winter, S: Summer

RQ1: How we analyse change impact on multi-language program?

RQ2: What are the possible architectures of heterogeneously licensed system?

RQ3: What is the impact of source code change on license compatibility?

WCRE: Working Conference on Reverse Engineering.

ICSE: International Conference on Software Engineering.

TSE: Transactions on Software Engineering.

JSME: Journal of Software Maintenance and Evolution: Research and Practice.

ICSM: International Conference on Software Maintenance.

6.2 Conclusion

To increase the productivity of developers during software development process, the software engineeres tend to reuse existing programs. The availability of Open Source Software (OSS) and proprietary system with open APIs amplified this activity of reusing programs. The reused programs can have different licenses and written with different programming language. Consequently, we are facing heterogeneous programs: Multi-language and heterogeneous licenses. Such heterogeneous program are difficult to analyse. In this context, we propose change impact analysis of multi-language programs and extend then to assess legal impact. Also, we want to suggest the possible architectures of a program when we combine different languages and licenses that can be conflicting. In this research proposal, we proposed an approach to reach our goal and a planning that we will guide us to achieve our goal.

Chapter 7

Annexe: Interaction between Perl/C and Java/Perl

7.0.1 Interaction between Perl and C

7.0.1.1 Calling C from Perl

Linking If you want to use C source code (or a C library) from Perl, you need to create a library that can be either dynamically loaded or statically linked into your perl executable (Dynamic loading is usually preferred, to minimize the number of different perl executables sitting around being different.). The glue code usually contains two files: a module file in Perl with .pm extension and a c file. To create the glue code there are two principal method. The first method is generating the library from .xs file using XSubpp tool or generating the library from Interface file(.i) using SWIG see Figure 7.1 :

1. Using XS
   You create the library by creating an XS file (.xs) containing a series of wrapper subroutines. The wrapper subroutines are not Perl subroutines, however; they are in the XS language, and we call such a subroutine an XSUB, for "eXternal SUBroutine". An XSUB can wrap a C function from an external library, a C function elsewhere in the XS file, or naked C code in the XSUB itself. You then use the xsubpp utility bundled with Perl to take the XS file and translate it into C code that can be compiled into a library that Perl will understand. But we can directly write the C code and linking it into your Perl executable. However, this would be tedious, especially if you need to write glue for multiple C functions, or if you're not familiar with the Perl stack discipline and other arcana. XS lets you write

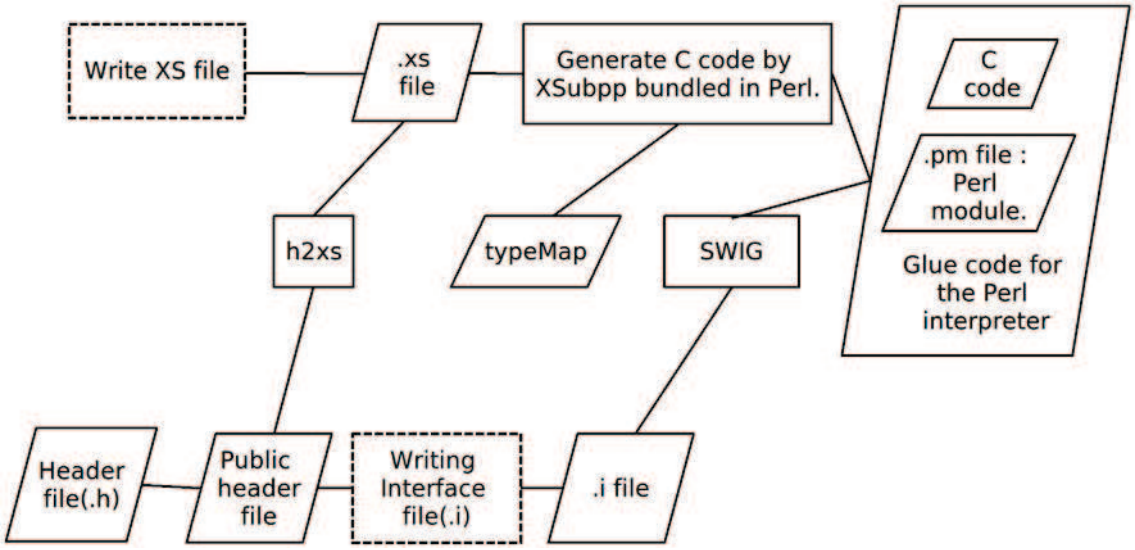


FIGURE 7.1: Process to call C code in Perl program

a concise description of what should be done by the glue, and the XS compiler xsubpp handles the rest.

We can also automate all the process by generating the xs file with h2xs. h2xs understands C header files (but not C++) and converts all constants and function prototypes to a meta language called XS. But a function declaration may still be too complex for scripting purposes, so this approach expects you to twiddle with the .xs file produced by h2xs and take the necessary steps to simplify the interface. Of course, the hand conversion is unnecessary if the interface is already simple enough.

Assuming your operating system supports dynamic linking, the end result will be a Perl module that behaves like any other module written in 100% pure Perl, but runs compiled C code under the hood. It does this by pulling arguments from Perl's argument stack, converting the Perl values to the formats expected by a particular C function (specified through an XSUB declaration), calling the C function, and finally transferring the return values of the C function back to Perl. These return values may be passed back to Perl either by putting them on the Perl stack or by modifying the arguments supplied from the Perl side.
[38]

An XS file begins with any C code you want to include, which will often be nothing more than a set of #include directives. After a MODULE keyword, the remainder of the file should be in the XS "language", a combination of XS directives and

XSUB definitions. We'll see an example of an entire XS file soon, but in the meantime here is a simple XSUB definition that allows a Perl program to access a C library function called sin(3). The XSUB specifies the return type (a double length floating-point number), the function name and argument list (with one argument dubbed x), and the type of the argument (another double):

```
double
sin(x)
...
double x
```

Each section of an XSUB starts with a keyword followed by a colon, such as INIT: or CLEANUP:. However, the first two lines of an XSUB always contain the same data: a description of the return type and the name of the function and its parameters. Whatever immediately follows these is considered to be an INPUT: section unless explicitly marked with another keyword.

The xsubpp program also needs to know how to convert from Perl's data types to C's data types. Often it can guess, but with user-defined types you may need to help it out by specifying the conversion in a typemap file. The default conversions are stored in PATH-TO-PERLLIB/ExtUtils/typemap. The typemap is split into three sections. The first section, labeled TYPEMAP, tells the compiler which of the code fragments in the following two sections should be used to map between C types and Perl values. The second section, INPUT, contains C code specifying how Perl values should be converted to C types. The third section, OUTPUT, contains C code specifying how to translate C types into Perl values.

The Mytest.xs file contains the XSUBs that tell Perl how to pass data to the compiled C routines. Initially, Mytest.xs will look something like this:

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
MODULE = Mytest PACKAGE = Mytest
```

Let's edit the XS file by adding this to the end of the file:

```
void
hello()
CODE:
printf(" Hello, world! ");
```

2. Using SWIG

   The SWIG system automatically generates simple XSUBs. SWIG (Simplified Wrapper and Interface Generator) is a freely available tool that integrates Perl, Python, Tcl, and other scripting languages with programs written in C, C++, and Objective-C [39]. SWIG, a tool designed to integrate C code with a variety of scripting languages including Perl, Python, and Tcl.

   SWIG is a specialized compiler that transforms ANSI C/C++ declarations into scripting language extension wrappers. While somewhat similar to h2xs, SWIG has a number of notable differences. First, SWIG is much less internals oriented than XS. In other words, SWIG interfaces can usually be constructed without any knowledge of Perl's internal operation. Second, SWIG is designed to be extensible and general purpose. Currently, wrappers can be generated for Perl, Python, Tcl, and Guile [39].

   suppose that you wanted to build a Perl interface to Thomas Boutell's gd graphics library. Since gd is a C library, images are normally created by writing C code such as follows [39]:

```
#include "gd.h"
int main() {
gdImagePtr  im;
FILE       *out;
int         blk,wht;
/* Create an image */
im=gdImageCreate(200,200);
/* Allocate some colors */
blk=gdImageColorAllocate(im,0,0,0);
wht=gdImageColorAllocate(im,255,255,255);
/* Draw a line */
gdImageLine(im,20,50,180,140,wht);
/* Output the image */
out=fopen("test.gif","wb");
gdImageGif(im,out);
fclose(out);
```

```
/* Clean up */
gdImageDestroy(im);
}
```

We How to write a similar code in Perl. Thus, the functionality of the gd must be exposed to the Perl interpreter. This is provided by SWIG interface :

```
// File : gd.i
%module gd
%{
#include "gd.h"
%}

typedef gdImage *gdImagePtr;

gdImagePtr gdImageCreate(int sx, int sy);
void       gdImageDestroy(gdImagePtr im);
void       gdImageLine(gdImagePtr im,
                       int x1, int y1,
                       int x2, int y2,
                       int color);

int   gdImageColorAllocate(gdImagePtr im,
                       int r, int g, int b);
void  gdImageGif(gdImagePtr im, FILE *out);

// File I/O functions (explained shortly)
FILE *fopen(char *name, char *mode);
void  fclose(FILE *);
```

In this file the function that we want to access from Perl are listed plus some SWIG directives which are preceeded by "%". The "%module". The %, % block is used to insert literal code into the output wrapper file. In this case, we simply include the "gd.h" header file. Finally, a few file I/O functions also appear. While not part of gd, these functions are needed to manufacture file handles used by several gd functions.

To run SWIG, the following command is executed:

```
unix > swig -perl5 gd.i
```

It generate wrappers for Perl 5

This produces two files, gd_wrap.c and gd.pm. The first file contains C wrapper functions that appear similar to the output that would have been generated by xsubpp. The .pm file contains supporting Perl code needed to load and use the module.

To build the module, the wrapper file is compiled and linked into a shared library. This process varies on every machine (consult the man pages), but the following steps are performed on Linux:

```
unix > gcc -fpic -c gd_wrap.c \
    -Dbool=char \
 -I/usr/lib/perl5/i586-linux/5.004/CORE
    unix > gcc -shared gd_wrap.o -lgd -o gd.so
```

At this point, the module is ready to use. For example, the earlier C program can be directly translated into the following Perl script:

```
    #!/usr/bin/perl
    use gd;
    # Create an image
    $im = gd::gdImageCreate(200,200);
    # Allocate some colors
    $blk=gd::gdImageColorAllocate($im,0,0,0);
    $wht=gd::gdImageColorAllocate($im,255,255,255);
    # Draw a line
    gd::gdImageLine($im,20,50,180,140,$wht);
    # Output the image
    $out=gd::fopen("test.gif","wb");
    gd::gdImageGif($im,$out);
    gd::fclose($out);
    \# Clean up
    gd::gdImageDestroy($im);
```

Fork : system call It is possible to do system call from Perl program as follow [40][**?**]:

- Solution 1: system call
  You can call any program from the command line using a system call. This is only useful if you do not need to capture the output of the program.

```
#!/usr/bin/perl
use strict;
use warnings;
my $status = system("C_Prog.exe");
```

  You'll need to bitshift the return value by 8 (or divide by 256) to get the return value of the program called:

```
#!/usr/bin/perl
use strict;
use warnings;
my $status = system("C_Prog");
if (($status >=8) != 0) {
 die "Failed to run vi";
}
```

- Solution 2: qx call
  If you need to capture the output of the program, use qx.

```
#!/usr/bin/perl
use strict;
use warnings;
my $info = qx(C_Prog);
print "C_Prog is: $info";
```

  Or if the output has multiple lines (e.g. the output of the "who" command can consist of many lines of data):

```
#!/usr/bin/perl
use strict;
```

```
use warnings;
my @info = qx(Perl.exe);
foreach my $i (@info) {
 int "$i is online";
}
```

You can also use backticks (‘) to achieve the same thing:

```
\#!/usr/bin/perl
use strict;
use warnings;
my @info = ‘Perl.exe‘;
foreach my \$i (@info) {
 print "$i is online";
}
```

SubClass It is not possible to inherit C class in Perl program.

7.0.1.2 IPC

Perl can use following IPC techniques of communication : Signals, Files, Pipes,System V IPC, Sockets.
C can use Signals, Pipes, Messaging Queue, Semaphores, Shared memory, Socket.
So Perl program can communicate with C program via Signals, Pipes, Sockets.

Plugin As we found a software XChatOSD [41] aimed on displaing XChat messages and it is written in Perl and have Plugins written in C, it is possible to write C plugin for Perl program.

7.0.1.3 Calling Perl from C

Linking Perl is itself written in C; the perl library is the collection of compiled C programs that were used to create your perl executable(/usr/bin/perl or equivalent). When you use Perl from C, your C program will–usually-allocate, ”run”, and deallocate a PerlInterpreter object, which is defined by the Perl library.

- Calling a Perl subroutine from your C program
  To call individual Perl subroutines, you can use any of the call_* functions documented in perlcall. In this example we'll use call_argv [42][43].

  That's shown below, in a program we will call showtime.c [43].

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

int main(int argc, char **argv, char **env)
{
    char *args[] = { NULL };
    PERL_SYS_INIT3(&argc,&argv,&env);
    my_perl = perl_alloc();
    perl_construct(my_perl);

    perl_parse(my_perl, NULL, argc, argv, NULL);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;

    /*** skipping perl_run() ***/

    call_argv("showtime", G_DISCARD | G_NOARGS, args);

    perl_destruct(my_perl);
    perl_free(my_perl);
    PERL_SYS_TERM();
}
```

  where showtime is a Perl subroutine that takes no arguments (that's the G_NOARGS) and for which I'll ignore the return value (that's the G_DISCARD). Those flags, and others, are discussed in perlcall.

I'll define the showtime subroutine in a file called showtime.pl:

```
print "I shan't be printed.";

sub showtime {
    print time;
}
```

Now compile and run:

```
% cc -o showtime showtime.c `perl -MExtUtils::Embed -e ccopts -e ldopts`

% showtime showtime.pl
818284590
```

In this particular case we don't have to call perl_run, as we set the PL_exit_flag PERL_EXIT_DESTRUCT_END which executes END blocks in perl_destruct.

If you want to pass arguments to the Perl subroutine, you can add strings to the NULL-terminated args list passed to call_argv. For other data types, or to examine return values, you'll need to manipulate the Perl stack.

Fork : Calling a Perl executable in C program

1. Adding a Perl Interpreter to the C program [44]: It is possible to execute a Perl script by adding a Perl Interpreter to the C program:

   A demonstration of embedding can be found in the file miniperlmain.c, included with the Perl source code. Here's a nonportable version of miniperlmain.c containing the essentials of embedding:

```
#include <EXTERN.h>               /* from the Perl distribution     */
#include <perl.h>                 /* from the Perl distribution     */

static PerlInterpreter *my_perl;  /***    The Perl interpreter    ***/

int main(int argc, char **argv, char **env)
{
```

```
    my_perl = perl_alloc();
    perl_construct(my_perl);
    perl_parse(my_perl, NULL, argc, argv, (char **)NULL);
    perl_run(my_perl);
    perl_destruct(my_perl);
    perl_free(my_perl);
}
```

When this is compiled with the command line above, you'll be able to use interp just like a regular Perl interpreter:

```
% interp -e "printf('%x', 3735928559)"
deadbeef
```

You can also execute Perl statements stored in a file by placing the filename in *argv*[1] before calling *perl_run*.

2. using system call [44]: You can use the "system" function.

   CPP / C++ / C Code:

   ```
   //Execute command.
   int result;
   result = system(your_command);
   ```

   "your_command" is the name of the command / filename you wish to execute. we can call a Perl executable file by :

   ```
   result = system("d:\\test.exe");
   ```

7.0.1.4 SubClass

IPC Perl can use following IPC techniques of communication : Signals, Files, Pipes,System V IPC, Sockets.
C can use Signals, Pipes, Messaging Queue, Semaphores, Shared memory, Socket.
So C program can communicate with Perl program via Signals, Pipes, Sockets.

Plugin

7.0.2 Interaction between Java and Perl

7.0.2.1 Calling Java From Perl

Linking

1. Java in Perl: Simple Constructors [45][46]
   Use *JPL::Class* to load the class:

   ```
   use JPL::Class "java::awt::Frame";
   ```

   Invoke the constructor to create an instance of the class:

   ```
   my $f = java::awt::Frame->new;
   ```

   You've got a reference to a Java object in $f, a Perl scalar.

2. Constructors that take parameters [45][46]
   If the constructor has parameters, look up the method signature with *getmeth*:

   ```
   my $new = getmeth("new", ['java.lang.String'], []);
   ```

   The first argument to getmeth is the name of the method. The second argument is a reference to an array that contains a list of the argument types. The final argument to getmeth is a reference to an array containing a single element with the return type. Constructors always have a null (void) return type, even though they return an instance of an object.

   Invoke the method through the variable you created:

   ```
   my $f = java::awt::Frame->$new( "Frame Demo" );
   ```

   The getmeth function is not just for constructors. You can use it to look up method signatures for any method that takes arguments.

Fork

- Solution 1: system call [40][**?**]
  You can call any program like you would from the command line using a system call. This is only useful if you do not need to capture the output of the program.

```
#!/usr/bin/perl
use strict;
use warnings;
my $status = system("JavaProgram.exe");
if (($status >=8) != 0) {
     die "Failed to run vi";
}
```

You'll need to bitshift the return value by 8 (or divide by 256) to get the return value of the program called:

- Solution 2: qx call [40][**?**]
  If you need to capture the output of the program, use qx.

```
#!/usr/bin/perl
use strict;
use warnings;
my $info = qx(JavaProgram.exe);
print "JavaProgram is: $info";
```

Or if the output has multiple lines (e.g. the output of the "who" command can consist of many lines of data):

```
#!/usr/bin/perl
use strict;
use warnings;
my @info = qx(JavaProgram.exe);
foreach my $i (@info) {
 print "$i is online";
}
```

You can also use backticks (‘) to achieve the same thing [40][**?**]:

```
#!/usr/bin/perl
use strict;
use warnings;
my @info = ‘JavaProgram.exe‘;
foreach my $i (@info) {
 print "$i is online";
}
```

IPC Perl can use following IPC techniques of communication : Signals, Files, Pipes,System V IPC, Sockets.
Java can use Signals, Semaphore, Pipes, Shared Memory, Domain Socket, RPC(remoting), Socket(UDP or TCP)
So Java can communicate with Perl program by using Signals, Pipes, Shared Memory, Socket.

Plugin

7.0.3 Calling Perl From Java

Linking Well-supported by JPL, but it is a complicated process [46]:

- The JPL preprocessor parses the .jpl file and generates C code wrappers for Perl methods. It also generates Java and Perl source files.
- The C compiler compiles the wrapper and links it to the libPerlInterpreter.so shared library, producing a shared library for the wrapper.
- The Java compiler compiles the Java source file, which uses native methods to load the wrapper.
- The wrapper connects the Java code to the Perl code in the Perl source file.

Fortunately, a generic Makefile.PL simplifies the process. This is a Perl script that generates a Makefile.
You can put Perl methods in your .jpl file. Perl methods are declared perl and use double curly braces to make life easier on the JPL preprocessor:

```
perl int perlMultiply(int a, int b) {{
my $result = $a * $b;
return $result;
}}
```

In your Java code, you can invoke Perl methods like a Java method. The native code wrappers take care of running the Perl code:

```
public void invokePerlFunction() {
    int x = 3;
```

```
        int y = 6;
        int retval = perlMultiply(x, y);
        System.out.println(x + " * " + y + " = " + retval);
    }
```

class MethodDemo :

```
class MethodDemo
    class MethodDemo {
     A Perl method to multiply two numbers and
        return the result.

    perl int perlMultiply(int a, int b) {{
        my $result = $a * $b;
        return $result;
    }}

//A Java method to call the Perl function.

    public void invokePerlFunction() {
        int x = 3;
        int y = 6;
        int retval = perlMultiply(x, y);
        System.out.println(x +" * "+ y +" = "+ retval);
    }

    public static void main(String[] args) {
        MethodDemo demo = new MethodDemo();
        demo.invokePerlFunction();
        }
}
```

IPC Perl can use following IPC techniques of communication : Signals, Files, Pipes,System V IPC, Sockets.
Java can use Signals, Semaphore, Pipes, Shared Memory, Domain Socket, RPC(remoting), Socket(UDP or TCP)
So Java can communicate with Perl program by using Signals, Pipes, Shared Memory, Socket.

Plugin

Fork If we want to call externel programs(executable program) in a Java application, we can use system call by creating Runtime Object and attaching it to system process [47].

Example:

```
import java.io.*;

public class Main {
       public static void main(String args[]) {
            try {
                Runtime rt = Runtime.getRuntime();
                Process pr = rt.exec("PerlScript.pl");
                BufferedReader input = new BufferedReader(new InputStreamReader(pr.getIn
                String line=null;
                while((line=input.readLine()) != null) {
                    System.out.println(line);
                }
                int exitVal = pr.waitFor();
                System.out.println("Exited with error code "+exitVal);
            } catch(Exception e) {
                System.out.println(e.toString());
                e.printStackTrace();
            }
        }
}
```

Method waitFor() will make the current thread to wait until the external program finish and return the exit value to the waited thread.

Bibliography

[1] Bernt Kullbach, Andreas Winter, Peter Dahm, and Jürgen Ebert. Program comprehension in multi-language systems. In *WCRE '98: Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, page 135, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8967-6.

[2] Extending perl:a first course. Last access: 31 August 2010. URL http://docstore.mik.ua/orelly/perl/advprog/ch18_01.htm.

[3] Andrea Capiluppi, Patricia Lago, and Maurizio Morisio. Characteristics of open source projects. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, page 317, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1902-4.

[4] Lawrence Rosen. *Open Source Licensing Software Freedom and Intellectual Property Law*. Prentice Hall, Juillet 2004. URL http://www.rosenlaw.com/oslbook.htm.

[5] Open source licenses by category. Last access: 31 August 2010. URL http://www.opensource.org/licenses/category.

[6] Can i apply the gpl when writing a plug-in for a non-free program? Last access: 31 August 2010. URL http://www.gnu.org/licenses/gpl-faq.html#GPLPluginsInNF.

[7] Apache license. Last access: 31 August 2010, . URL http://en.wikipedia.org/wiki/Apache_License.

[8] License. Last access: 31 August 2010, . URL http://www.apache.org/licenses/.

[9] Standing against license proliferation. Last access: 31 August 2010. URL http://google-opensource.blogspot.com/2008/05/standing-against-license-proliferation

[10] Daniel M. German, Massimiliano Di Penta, and Julius Davies. Understanding and auditing the licensing of open source software distributions. *International Conference on Program Comprehension*, 0:84–93, 2010. ISSN 1063-6897. doi: http://doi.ieeecomputersociety.org/10.1109/ICPC.2010.48.

[11] Shawn A. Bohner. Software change impact analysis. 1996.

[12] Giovanni Beltrame, Luca Fossati, and Donatella Sciuto. Resp: a nonintrusive transaction-level reflective mpsoc simulation platform for design space exploration. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(12):1857–1869, 2009. ISSN 0278-0070. doi: http://dx.doi.org/10.1109/TCAD.2009.2030268.

[13] Daniela Carneiro da Cruz. *Methods and techniques to analyze multi-level code to explore software components*. PhD thesis, Universidade do minho, 2008.

[14] Daniel M. German and Ahmed E. Hassan. License integration patterns: Addressing license mismatches in component-based development. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 188–198, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: http://dx.doi.org/10.1109/ICSE.2009.5070520.

[15] Klaas-Jan Stol and Muhammad Ali Babar. Challenges in using open source software in product development: a review of the literature. In *FLOSS '10: Proceedings of the 3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, pages 17–22, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-978-7. doi: http://doi.acm.org/10.1145/1833272.1833276.

[16] Mitch Bayersdorfer. Managing a project with open source components. *interactions*, 14(6):33–34, 2007. ISSN 1072-5520. doi: http://doi.acm.org/10.1145/1300655.1300677.

[17] R. C. Osterberg. *Substantial Similarity in Copyright Law*. Practising Law Institute, 2003. URL http://openlibrary.org/books/OL3698208M/Substantial_similarity_in_copyright_law.

[18] Z. Obrenovic and D. Gasevic. Open source software: All you do is put it together. *Software, IEEE*, 24(5):86–95, sep. 2007. ISSN 0740-7459. doi: 10.1109/MS.2007.141.

[19] Panagiotis K. Linos. Polycare: a tool for re-engineering multi-language program integrations. In *ICECCS*, pages 338–, 1995.

[20] Panagiotis K. Linos, Zhi hong Chen, Seth Berrier, and Brian O'Rourke. A tool for understanding multi-language program dependencies. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 64, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1883-4.

[21] Panos Linos, Whitney Lucas, Sig Myers, and Ezekiel Maier. A metrics tool for multi-language software. In *SEA '07: Proceedings of the 11th IASTED International*

Conference on Software Engineering and Applications, pages 324–329, Anaheim, CA, USA, 2007. ACTA Press. ISBN 978-0-88986-706-2.

[22] Daniel L. Moise and Kenny Wong. Extracting facts from perl code. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 243–252, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2719-1. doi: http://dx.doi.org/10.1109/WCRE.2006.28.

[23] Daniel L. Moise, Kenny Wong, H. James Hoover, and Daqing Hou. Reverse engineering scripting language extensions. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 295–306, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2601-2. doi: http://dx.doi.org/10.1109/ICPC.2006.42.

[24] Daniel L. Moise and Kenny Wong. Extracting and representing cross-language dependencies in diverse software systems. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 209–218, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2474-5. doi: http://dx.doi.org/10.1109/WCRE.2005.19.

[25] Laurent Deruelle and Henri Basson. An eclipse platform for analysis and manipulation of distributed multi-language software. In *CAINE*, pages 100–105, 2008.

[26] Ahmed E. Hassan and Richard C. Holt. Architecture recovery of web applications. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 349–359, New York, NY, USA, 2002. ACM. ISBN 1-58113-472-X. doi: http://doi.acm.org/10.1145/581339.581383.

[27] Thomas A. Alspaugh, Hazeline U. Asuncion, and Walt Scacchi. Intellectual property rights requirements for heterogeneously-licensed systems. In *RE '09: Proceedings of the 2009 17th IEEE International Requirements Engineering Conference, RE*, pages 24–33, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3761-0. doi: http://dx.doi.org/10.1109/RE.2009.22.

[28] Timo Tuunanen, Jussi Koskinen, and Tommi Kärkkäinen. Automated software license analysis. *Automated Software Engg.*, 16(3-4):455–490, 2009. ISSN 0928-8910. doi: http://dx.doi.org/10.1007/s10515-009-0054-z.

[29] Leon Moonen. Generating robust parsers using island grammars. *Reverse Engineering, Working Conference on*, 0:13, 2001. ISSN 1095-1350. doi: http://doi.ieeecomputersociety.org/10.1109/WCRE.2001.957806.

[30] Yann-Gaël Guéhéneuc. Ptidej: A flexible reverse engineering tool suite. In *ICSM*, pages 529–530. IEEE, 2007. URL http://dblp.uni-trier.de/db/conf/icsm/icsm2007.html#Gueheneuc07.

[31] G. Antoniol, G. Casazza, M. Di Penta, and R. Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59(2):181–196, 2001. ISSN 0164-1212. doi: DOI:10.1016/S0164-1212(01)00061-9. URL http://www.sciencedirect.com/science/article/B6V0N-449TJ06-J/2/1194eca49fa9a9d8dbafd

[32] How to run command-line or execute external application from java. Last access: 31 August 2010, . URL http://www.linglom.com/2007/06/06/how-to-run-command-line-or-execute-external-appli

[33] Embed java code into your native apps. Last access: 31 August 2010. URL http://www.javaworld.com/javaworld/jw-05-2001/jw-0511-legacy.html.

[34] Collectd. Last access: 31 August 2010. URL http://collectd.org/features.shtml.

[35] Stephen H. Kaisler. *Software Paradigms*. John Wiley & Sons, 2005. ISBN 0471483478.

[36] Daniel M. German, Y. Manabe, and K. Inoue. A sentence-matching method for automatic license identification of source code files. Under review, 2009. URL http://turingmachine/~dmg/papers/.

[37] Martin Pinzger and Harald Gall. Pattern-supported architecture recovery. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, page 53, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1495-2.

[38] Extending perl (using c from perl). Last access: 31 August 2010, . URL http://docstore.mik.ua/orelly/perl3/prog/.

[39] David M. Beazley, David Fletcher, Dominique Dumont, and Hewlett Packard. Perl extension building with swig. In *in O'Reilly Perl Conference 2.0*, 1998.

[40] Perl programming documentation. Last access: 31 August 2010, . URL http://perldoc.perl.org/functions/syscall.html.

[41] Xchatosd perl script / c plugin files. Last access: 31 August 2010, . URL http://xchatosd.sourceforge.net/.

[42] John Moreland. Introduction to perl. Last access: 31 August 2010. URL http://www.sdsc.edu/~moreland/courses/IntroPerl/docs/manual/pod/perlipc.html.

[43] Jon Orwant and Doug MacEachern. perlembed - how to embed perl in your c program. Last access: 31 August 2010. URL http://search.cpan.org/~jesse/perl-5.12.1/pod/perlembed.pod.

[44] Larry Wall. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000. ISBN 0596000278.

[45] Calling java from perl. Last access: 31 August 2010, . URL http://perl.active-venture.com/jpl/docs/Tutorial.html.

[46] Tutorial - perl and java. Last access: 31 August 2010, . URL http://sunsite.ualberta.ca/Documentation/Misc/perl-5.6.1/jpl/docs/Tutorial.html.

[47] Velocity reviews - java - execute perl in java(forum). Last access: 31 August 2010, . URL http://www.velocityreviews.com/forums/t129578-execute-perl-in-java.html.

L'École Polytechnique se spécialise dans la formation d'ingénieurs et la recherche en ingénierie depuis 1873

École Polytechnique de Montréal

École affiliée à l'Université de Montréal

Campus de l'Université de Montréal
C.P. 6079, succ. Centre-ville
Montréal (Québec)
Canada H3C 3A7

www.polymtl.ca

ÉCOLE
POLYTECHNIQUE
MONTRÉAL