REUSE MINER: MINING FRAMEWORK INSTANTIATION PROCESSES

Talita Lopes Gomes

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Toacy Cavalcante de Oliveira

Rio de Janeiro
Junho de 2015

REUSE MINER: MINING FRAMEWORK INSTANTIATION PROCESSES

Talita Lopes Gomes

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

_____

Prof. Geraldo Bonorino Xexéo, D.Sc.

_____

Prof. Claudia Maria Lima Werner, D.Sc.

_____

Prof. Paulo Cesar Masiero, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

JUNHO DE 2015

Gomes, Talita Lopes

Reuse Miner: Mining Framework Instantiation Processes / Talita Lopes Gomes – Rio de Janeiro: UFRJ/COPPE, 2015.

XVII, 109 p.: il.; 29,7 cm.

Orientador: Toacy Cavalcante de Oliveira

Dissertação (Mestrado) – UFRJ/ COPPE/ Programa de Engenharia de Sistemas e Computação, 2015.

Referências Bibliográficas: p. 97-102.

1. Frameworks orientados a objetos. 2. Reutilização de Software. 3. Mineração de Processos. 4. RDL. I. Oliveira, Toacy Cavalcante de. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Acknowledgements

Firstly, I would like to thank my mother Célia, my stepfather Antonio Carlos and my boyfriend Rafael for their support, comprehension and motivation throughout the development of this work. I'd like to thank my brother and my father as well, whom I know are always cheering for me.

I thank my advisor Toacy for his support, patience and for our always interesting talks about academic life and "the outside world". I also thank him for giving me the opportunity to visit and cooperate with the Computer Science research group of University of Waterloo, where the basis for this work was developed. The interaction with teachers and students, either visitors or not, was a very rich experience, personally and professionally, and I am very glad that I have had such amazing opportunity.

I also thank all my teachers that contributed to my education. Despite all of its problems, UFRJ is a university with a lot to offer and I consider myself lucky for having the chance to enjoy some of it.

Last but not least, I want to thank all my friends, who always give me support and are always willing to offer help when needed.

Agradecimentos

Em primeiro lugar, agradeço à minha mãe Célia, ao meu padrasto Antonio Carlos e ao meu namorado Rafael por todo apoio, compreensão e motivação ao longo do mestrado. Agradeço também ao meu irmão e meu pai, que mesmo participando tendo participado um pouco menos, sei que estão sempre na torcida.

Ao meu orientador Toacy obrigada pelo apoio, paciência e pelos conversas sempre interessantes sobre a vida acadêmica e a vida “lá fora”. Agradeço também por ter me proporcionado a possibilidade de visitar e colaborar com o grupo de pesquisa de Ciência da Computação da Universidade de Waterloo, onde a base deste trabalho foi desenvolvida. A convivência com os professores e alunos visitantes ou não foi uma experiência muito enriquecedora, tanto pessoalmente como profissionalmente, e fico muito feliz de ter tido a oportunidade de vivenciar isso.

Agradeço também a todos os professores que contribuíram com a minha formação acadêmica desde a graduação. Apesar de seus problemas, a UFRJ é uma faculdade que tem muito a oferecer e me considero uma pessoa de sorte por ter tido a chance de aproveitar um pouco do que ela tem a oferecer.

Por último, mas igualmente importante, gostaria de agradecer aos meus amigos, que sempre me dão apoio e estão dispostos a oferecer uma ajuda se preciso for.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

REUSE MINER: MINERANDO PROCESSOS DE INSTANCIAÇÃO PARA FRAMEWORKS

Talita Lopes Gomes

Junho/2015

Orientador: Toacy Cavalcante de Oliveira

Programa: Engenharia de Sistemas e Computação

Reutilização de software é uma parte essencial do desenvolvimento de software cujo objetivo principal é obter melhorias na qualidade do produto final juntamente com um tempo de desenvolvimento reduzido. Uma das tecnologias disponíveis para a reutilização de software considerando aplicações de certo domínio são os frameworks orientados a objetos. Estes frameworks possuem duas partes principais: um cerne imutável, compartilhado por todas as aplicações feitas a partir deste framemork; e interfaces customizáveis – *hotspots* –, usadas pelos desenvolvedores para incluir aspectos específicos de suas aplicações. Devido à esta flexibilidade, frameworks possuem uma estrutura complexa que pode impactar sua curva de aprendizado. É de extrema importância para o sucesso de um framework, que o desenvolvedor seja capaz de identificar e usar os *hotspots* de interesse.

Neste contexto, esta dissertação apresenta o Reuse Miner, uma abordagem para descobrir processos de instanciação de frameworks a partir do código de aplicações existentes. O processo descoberto contém *hotspots* e também informações sobre *hotspots* que são opcionais e sobre os que são repetíveis. Este processo é representado usando RDL (*Reuse Description Language*), o que tem dois objetivos principais: i) complementar a documentação do framework; ii) prover suporte semi-automático ao processo de instanciação com o uso da ReuseTool, ferramenta capaz de guiar o processo de instanciação a partir das informações em um processo RDL.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

REUSE MINER: MINING FRAMEWORK INSTANTIATION PROCESSES

Talita Lopes Gomes

June/2015

Advisor: Toacy Cavalcante de Oliveira

Department: Computer Science Engineering

Software reuse is an essential part of software development that targets improvements in the quality of the resulting software with reduced development time. Object-oriented frameworks are a software reuse technology that targets the reuse for applications of a certain domain, offering a non-customizable core implementation and flexible parts – hot spots – that should be augmented with developer's needs. To achieve such flexibility, frameworks have a complex structure which impacts on its understandability and, in turn, can impact in the learning curve. It is crucial for the framework success that developers can identify the hotspots of interest.

In this context, this dissertation proposes the Reuse Miner, an approach to mine instantiation processes from application code. The mined instantiation process contains not only the framework hotspots, but also information about what is optional and what is repeatable. This process is represented using RDL (Reuse Description Language) which has two main purposes: I) complement the framework documentation; ||) offer the possibility of semi-automating the instantiation process with the use of the ReuseTool, a tool to support framework instantiations based on the steps described in the RDL specification.

Table of Contents

INDEX OF FIGURES

INDEX OF TABLES

INDEX OF CODES

INDEX OF EQUATIONS

INDEX OF PROCESSES

LIST OF ABBREVIATIONS

BPMN – **B**usiness **P**rocess **M**odeling **N**otation

DSL – **D**omain **S**pecific **L**anguage

GEF – Graphical **E**diting **F**ramework

PAIS - **P**rocess-**A**ware **I**nformation **S**ystems

PST – **P**rogram **S**tructure **T**ree

RDL – **R**euse **D**escription **L**anguage

RPST – **R**efined **P**rogram **S**tructure **T**ree

XES - e**X**tensible **E**vent **S**tream

XPDL – **X**ML **P**rocess **D**efinition **L**anguage

1 Introduction

This chapter introduces this dissertation and presents its context and main goals. It also presents the organization of this document, exposing the main topics covered by next chapters.

1.1 Motivation

The reuse of previously developed pieces of software has always been part of software development (FRAKES & KANG, 2005). The emergence of object-oriented technology enhanced software reuse by introducing a number of qualities, including problem-orientation and domain analysis. Reusable assets in object-orientation vary in a continuum ranging from objects and classes to design patterns and object-oriented frameworks, where object-oriented frameworks represent the state-of-the-art (MATTSSON, 1996).

An object-oriented framework targets applications of a certain domain, putting together semi-complete software components that represent the basis for implementing applications for that domain. These components represent the compilation of prior domain knowledge and common solutions for requirements and challenges of that domain. Thus, the use of such frameworks may reduce development time, improve software quality, and improve developers' productivity (FAYAD & SCHMIDT, 1997).

Figure 1 illustrates two editors implemented on top of the Graphical Editing Framework (GEF) to exemplify the use of frameworks in the creation of custom domain applications (GEF, 2002). GEF targets the development of customized graphical editors, offering developers a set of classes, methods, and interfaces that can be used

to build custom editors. In Figure 1, two applications that are distributed with the framework are depicted: the Shapes editor (on the right) and the Logic editor (on the left). The former is a simple editor that allows users to create two different types of shapes (ellipses and rectangles) and to connect them using two different types of connections (solid or dashed). On the other hand, the Logic editor offers only one type of connection, but it provides a wider variety of elements to create logic circuits.

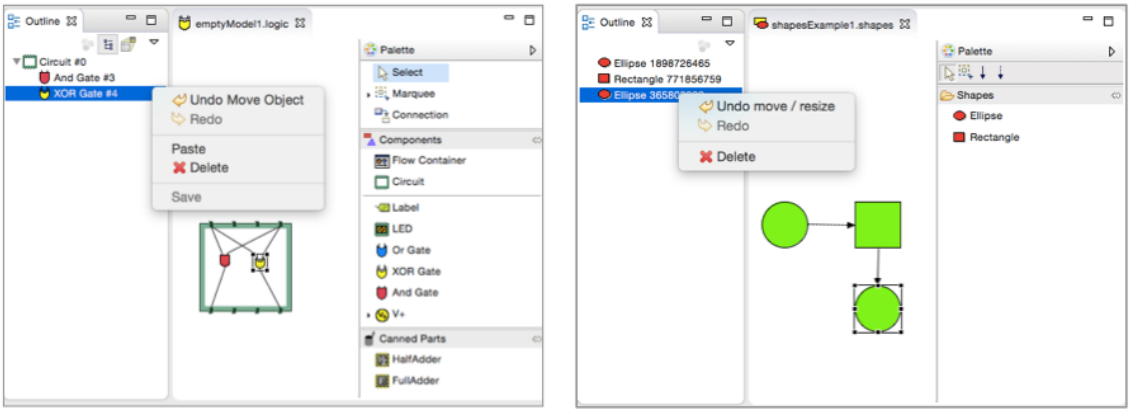


Figure 1. GEF example applications

Although the goal of each final editor is different, there are several similarities between them, including: both offer graphical elements that can be used to create models; graphical elements are organized in a Palette; the elements added to the model are listed in the Outline tab, which is located in the left side of the screen; and both have commands to select and delete elements in the model. All these similarities, among others, exist because they constitute flexible points that GEF exposes to create custom applications. These flexible points are known as *hotspots*, and they are designed to accommodate application-specific code.

The increment of hotspots with application code in order to create new domain-specific applications corresponds to a *framework instantiation*. To be able to handle a variety of application code, the framework design often includes complex constructs to accommodate these flexible points. While this flexibility is crucial to increase the

coverage of the framework, it also overloads the framework and makes it harder to understand. In fact, the identification and the adaptation of hotspots are major problems regarding the instantiation process (BRAGA and MASIERO, 2003).

In this context, good quality documentation is crucial for the success of a framework. The lack of documentation on how to reuse a framework may have a negative impact on the adoption of this framework (MATTSSON, 1996). Nevertheless, it is not trivial to maintain a high-quality and up-to-date documentation: the results of a survey conducted in (LETHBRIDGE *et al.*, 2003) indicate a slower update rate for the software documentation when compared to updates made to the software itself. Although the out-of-date documentation may remain useful (LETHBRIDGE *et al.*, 2003), previous work has shown that important information may be left out of the documentation (BRUCH *et al.*, 2010; THUMMALAPENTA & XIE, 2008). For instance, classes designed for inheritance must be well documented to avoid unexpected reuse, but this is not a simple task, since it depends on the definition of what should be exposed for subclasses or not (BLOCH, 2008).

There are several solutions in the literature that focus on ways to improve the documentation of frameworks in order to overcome the lack of framework learning resources. Among these works, we have the use of exemplars to help understand the framework architecture throughout classes' collaboration (GANGOPADHYAY and MITRA, 1995), the use of design patterns or pattern languages to describe the framework in terms of problems and their respective solutions (JOHNSON, 1992; BRAGA and MASIERO, 2004) and the use of cookbooks with a collection of recipes for solving different problems (ORTIGOSA and CAMPO 1999, OLIVEIRA et al. 2007).

Additionally, some of these solutions provide active guidance to the instantiation process based on the framework documentation (BRAGA and MASIERO 2003; ORTIGOSA and CAMPO 2000; OLIVEIRA, ALENCAR, and COWAN 2011). In this

case, the framework instantiation is automated or, more precisely, semi-automated since some interaction with the framework re-user during the process execution is needed. The instantiation is assisted by a tool that guides the developer throughout the necessary steps to create a new application based on an existing specification that describes these steps (SALVADOR, 2009). For this reason, the instantiation happens in a *process-oriented* manner, while the manual instantiation is *ad hoc*.

One of such tools is the ReuseTool (OLIVEIRA *et al.*, 2011). The ReuseTool assists the instantiation of frameworks based on a process described in RDL (Reuse Description Language), a domain specific language designed for the instantiation of frameworks. A RDL process contains all *reuse actions* that should be performed during the instantiation. A reuse action corresponds to the implementation of a framework hotspot through the extension of classes and methods extension, and/or the implementation of interfaces. The language's syntax also provides the commands to describe the process flow, such as the IF command for conditionals and the LOOP command to indicate the repetition of activities.

Currently, ReuseTool depends on the framework developer or on a specialist to specify the framework instantiation process in RDL, which represents an extra burden on the framework developers and limits ReuseTool adoption. An interesting alternative to developers would be to leverage on the knowledge embedded on the framework instances to discover the reuse process.

Based on this rationale, this dissertation presents Reuse Miner, an approach proposed to mine RDL processes from code repositories that contain instances of a given object-oriented framework. We want to investigate the use of information in software repositories to recover the instantiation process of a framework of interest. The objectives of this study are twofold. First, the discovered process can be the start point for the development of new applications, documenting the steps the framework

user should perform. Moreover, it can be used as input for the process automation, providing the information needed to guide the user throughout the process.

The remaining of this chapter introduces the main aspects of the proposed solution. In Section 1.2, an overview of the solution is presented, highlighting its main goals. Next, Section 1.3 outlines the entire dissertation, describing the topics addressed by following chapters.

1.2 Reuse Miner

Object-oriented frameworks assemble the knowledge and common solutions for problems of a certain domain. The underlying goal when using frameworks is to build several applications for the same domain with improved quality and reduced time and effort. Each new application corresponds to a framework instantiation and it involves modifying the framework flexible points to accommodate the developer's needs, as previously mentioned.

A framework instantiation can be described in terms of a process: the *instantiation process*. A process organizes activities so that they can be managed, measured, supported, and improved to reach certain goal (ESTUBLIER, 2006). Following this concept, an instantiation process defines the activities that should be performed with the goal of creating new framework applications. Each activity in this process corresponds to a framework extension activity, i.e., a class extension, a method extension, or the implementation of an interface. The execution of the process may involve the repetition of activities and choices, i.e., some reuse actions can be left out from the final application.

A process can be formally expressed in terms of a process model, which can be consistently repeated to reach the process goal. To this end, the process defines *how* a job must be performed and *what* are the resources involved. Once modeled, the

process model can be used to transfer knowledge to newcomers and for improvement purposes (ESTUBLIER, 2006). The modeling of the instantiation process, however, requires knowledge of how the framework works. In addition to the framework developer, the knowledge about how the framework works can also be obtained from existing framework applications. In this context, this dissertation describes an approach – *Reuse Miner* – that intends to discover an *executable* process for the instantiation of a target framework based on existing framework applications found in software repositories.

The use of data mining over software repositories allows us to gather information about several framework instantiations and to discover useful information that can be applied to new instantiations. These repositories make a plethora of data available with respect to the software development, which then creates singular opportunities to use data mining techniques for learning how to reuse a framework based on what has been done in the past. In the context of object-oriented frameworks, data mining has been applied to support a range of activities in the development of applications, such as the identification of hotspots from application code (THUMMALAPENTA & XIE 2008) and the recommendation of APIs (Zhong *et al.*, 2009; SILVA JUNIOR *et al.*, 2012; BRUCH, *et al.,* 2009). Reuse Miner tries to discover, with the support of *process discovery algorithms*, the underlying dependencies between reuse actions activities.

Similarly to the research field of mining software repositories, process mining gained attention with the growth of the amount of event data generated by executing business processes. Event data corresponds to all data generated by an information system during the execution of a business process (e.g.: purchase of flight tickets). This data is usually associated with an event log, storing multiple executions of a single process, and it can be applied to uncover the underlying process connecting the

executed activities; this approach is known as *process discovery*. In the absence of a modeled process, process discovery algorithms can be used to extract one from execution data (AALST, 2011).

Similarly to what happens in Process Mining for business, we want to discover a model for the instantiation process based on previous process execution. In our case, a process execution corresponds to a framework application and the activities on this process are equivalent to the reuse actions performed.

Moreover, Reuse Miner aims at discovering an *executable* process, i.e., we want the discovered process to provide guidance to the developer throughout the instantiation. To achieve this, the final outcome of Reuse Miner is an RDL process

It is possible to summarize the main goals of this dissertation in two, as follows:

- **G1. *Find an instantiation process*** → define the steps needed to instantiate a framework and their ordering relations;
- **G2. Allow the execution of the instantiation process** → A RDL (Reuse Description Language) process will be used to accomplish this goal.

Figure 2 gives an overview of the solution, indicating the main steps to achieve our goals: Log mining, Process mining and RDL generation. Figure 2.1 and 2.2 deal with goal G1, the process discovery phase. Figure 2.3 comply with goal G2, translating the discovered process to RDL (Reuse Description Language), thus allowing it to be executed by the ReuseTool.

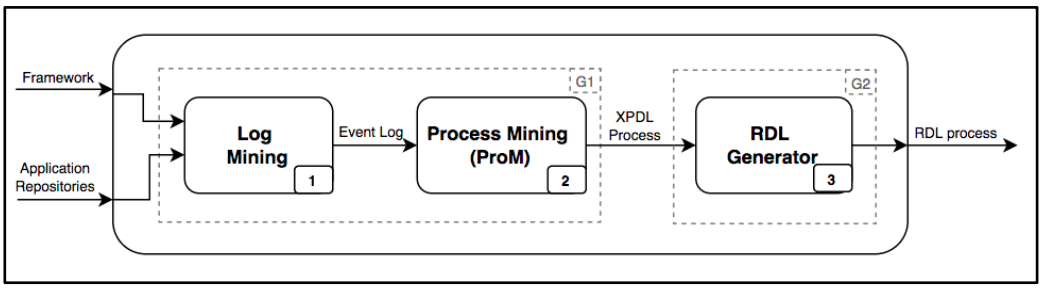


Figure 2. Solution overview

Each part of the solution exposed in Figure 2 plays an important part in the main solution and their purposes are stated below. All steps will be further explained in Chapter 4.

1. **Log Mining** → *Structure framework application data in an event log*. Process discovery algorithms expect as input an event log conforming to the process mining meta-model in (DONGEN, 2005). The log miner extracts data from code and structures it using event logs. The extraction of event logs must consider the correlation and ordering between events, as explained in Chapter 4, Section 4.2.
2. **Process Mining** → *Discover an instantiation process for the target framework.* The outcome of this stage is a business process represented using XPDL (XML Process Definition Language (WFMC, 2012)). This step is performed using ProM framework, a framework developed to foment the advances in process mining and it is better detailed in Chapter 4, Section 4.3.
3. **RDL Generation** → *The transformation of the XPDL process in an RDL script.* This step involves the identification of sequences, conditionals and loops in the BPMN (Business Process Modeling Notation) process. This identification is accomplished with the assistance of a RPST (Refined Program Structure Tree - (VANHATALO *et al.*, 2008)), as Chapter 4, Section 4.4 describes.

A great advantage of a process-oriented approach is the ability to provide guidance to the framework user while creating new applications. The execution of the RDL process by the ReuseTool generates an UML model for the application, tailored to the developer's needs. Moreover, the RDL process can also be used to complement an existing documentation, by presenting the hotspots most likely to be used, in the order they were previously used by other applications.

1.3 Organization

This work is organized in 6 chapters as follows. In this chapter, an introduction to the work was given pointing out the advantages of a process-oriented instantiation for object-oriented frameworks.

Chapter 2 gives the underlying concepts for this work, presenting the ReuseTool and RDL (Reuse Description Language). Chapter 2 also exposes the process mining discipline and its main elements. Process mining algorithms, more specifically process discovery algorithms, are a key element to the solution proposed in this dissertation. It presents the workflow to apply process discovery to business process, exposes the current standard for event logs and explains how two important process discovery algorithms work, the α-algorithm and the heuristic miner (AALST, 2011).

Chapter 3 presents the related work. The focus is on the use of data mining techniques to improve the instantiation of object-oriented frameworks. Different approaches can be found in the literature, such as the mining of code examples and the mining of framework hotspots.

Chapter 4 details the Reuse miner approach to discover instantiation processes for frameworks. This approach has three main phases: the mining of event logs from code repositories, the discovery of the instantiation process represented using BPMN and the subsequent representation of the mined process using RDL.

Chapter 5 presents the validation of the proposed solution along with threats to validity. Precision and recall metrics are used to evaluate the mined process.

Finally, Chapter 6 concludes this work, presenting its results and future work. Limitations of this work are also discussed.

2 Theoretical Foundation

This chapter presents the theoretical foundation for this dissertation. It gives an overview of the ReuseTool, a tool to assist the instantiation of object-oriented frameworks, and RDL, the language used to write reuse processes for the ReuseTool. Finally, the process mining discipline and its main elements are presented. Process discovery techniques will be used to mine reuse processes specification.

2.1 Object-oriented Frameworks

Software reuse is the reuse of existing software or software knowledge in the creation of new software with the ultimate purpose of improving software quality and developers' productivity. To achieve these goals, a key factor is domain engineering (FRAKES & KANG 2005), performed to gather knowledge and common solutions for problems of a target domain and make it available to the development of several applications. Object-oriented frameworks are a software technology that uses domain engineering aiming at improving software reuse.

Object-oriented frameworks are semi-complete software systems created to facilitate the development of several applications of a certain domain. As semi-complete systems, frameworks offer the reuse of both code and design. To accomplish this, a framework is composed of frozen spots and *hotspots*. The frozen spots are the immutable parts of the framework that constitute the framework kernel, and, as such, are reused in all framework applications. Hotspots correspond to the flexible parts of the framework, designed for customization. The extension of hotspots with applications' needs represent a *framework instantiation* (MARKIEWICZ & LUCENA, 2000).

Figure 3 gives an overview of a framework-centered software development process. Two stages can be identified in this process. The framework development occurs in the first stage and it involves the domain analysis and the definition of a framework design to meet the domain requirements. In this stage, the *development for reuse* takes place. In the second stage, the framework is actually reused in the implementation of a variety of applications. This corresponds to the *development with reuse*.

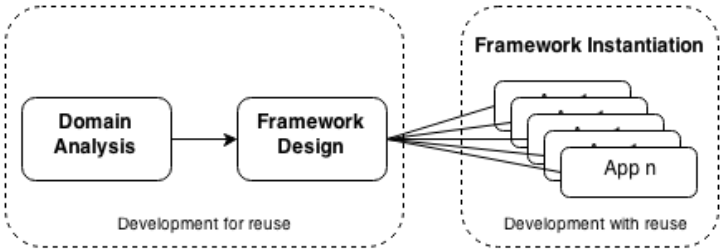


Figure 3. Framework-centered development. Adapted from (MARKIEWICZ & LUCENA, 2000)

The use of frameworks has the potential to decrease development time and effort. Nonetheless, one of the major challenges when starting to use a framework is understanding how to build new applications from it (KIRK *et al.*, 2007). The effort to build an application using the framework should not be greater than building the same application without using it. In this scenario, information on how to use the framework's hotspots is vital. Developers need a good documentation exposing the framework hotspots. In fact, the absence of documentation that eases the reuse of a framework may impact in the adoption of that framework (MATTSSON 1996).

Several types of documentation have been proposed in the literature in order to assist the instantiation of object-oriented frameworks. Johnson (1992) proposed the use of design patterns in a *cookbook-like document* as a documentation style for documenting framework features. The inspiration to use design patterns came from the architecture discipline where Christopher Alexander developed them to document how

to solve particular problems in architecture. These design patterns were designed for people with little or no training. Similarly, frameworks are designed for both experienced and novice developers and a cookbook describes a recurrent problem in the framework domain, including a description of the problem, a detailed discussion on how to solve the problem and a summary of the solution. The solution discussion may include examples and pointers to other framework patterns. A cookbook focuses on how to use the framework, but it can also bring useful information related to the framework design, i.e., how the framework works.

A limitation of cookbooks is the lack of automated guidance to the framework developer and the lack of formality for its content. A solution to mitigate these limitations is active guidance (PREE, 1995). Pree (1995) proposed the use of design books and active cookbooks to assist the reuse of frameworks. Design books document framework patterns in terms of metapatterns composed of hooks, templates and the interactions between them. Templates implement functionality in a generic way using generic parts called hooks – the framework hotspots. Active cookbooks offer implementation details for a framework adaptation, i.e., implementation of hooks methods. A tool guides the developer over the steps described in the cookbooks.

In (ORTIGOSA & CAMPO, 1999), the authors propose an approach called SmartBooks to support framework instantiation based on active cookbooks. The purpose of SmartBooks is to offer more flexibility to the instantiation process in comparison to active cookbooks with the use of an instantiation plan generated from instantiation rules. Instantiation plans based on functional requirements guide the developer throughout the functionality implementation. Instantiation rules need to be defined to generate these instantiation plans. Although, SmartBooks offer more flexibility than active cookbooks and have a functionality-centered approach, the need

for creating instantiation rules add an extra burden to the process (OLIVEIRA *et al.*, 2011).

Similarly to SmartBooks, *ReuseTool* (OLIVEIRA *et al.*, 2011) provides semi-automated assistance to the reuse of frameworks. In lieu of instantiation plans, the ReuseTool expect the framework reuse process to be represented in RDL (Reuse Description Language). Currently, ReuseTool depends on the manual specification of RDL, which may pose an obstacle to its adoption. In this scenario, the goal of Reuse Miner is to mine reuse processes specifications in RDL, gathering information from existing framework applications (GOMES *et al.*, 2014). It is a common practice to distribute example applications with the framework to support developers in the learning process. The underlying rationale of Reuse Miner is the same, using former applications to recover which hotspots should be used in the development of new applications. To achieve this purpose, it uses process discovery techniques from Process Mining discipline.

The remaining of this chapter discusses the theoretical foundation of Reuse Miner. Section 2.2 describes how the ReuseTool works. In addition, it exposes RDL in more details, presenting its main commands. Next, an introduction to the process mining discipline is given in Section 2.3, presenting aspects relevant to the Reuse Miner in the following sub-sections: i) Section 2.3.1 explains how data is structured in event logs to perform process mining; ii) Section 2.3.2 explains different types of process representation; iii) 2.3.3 explains two important process discovery algorithms (α-algorithms and the heuristic miner).

2.2 ReuseTool

Every framework instantiation is executed following a process, which can be either manual or tool-supported (SALVADOR, 2009). A Manual instantiation is performed

following the information provided by the framework documentation or using framework applications and examples as a guide. With the assistance of a tool, it is possible to guide developers through a sequence of steps defined by the framework engineer, thus following the same rationale used to create the framework and respecting framework constraints (OLIVEIRA *et al.*, 2011). ReuseTool is devoted to the second type of instantiation, providing active guidance to the reuse of frameworks.

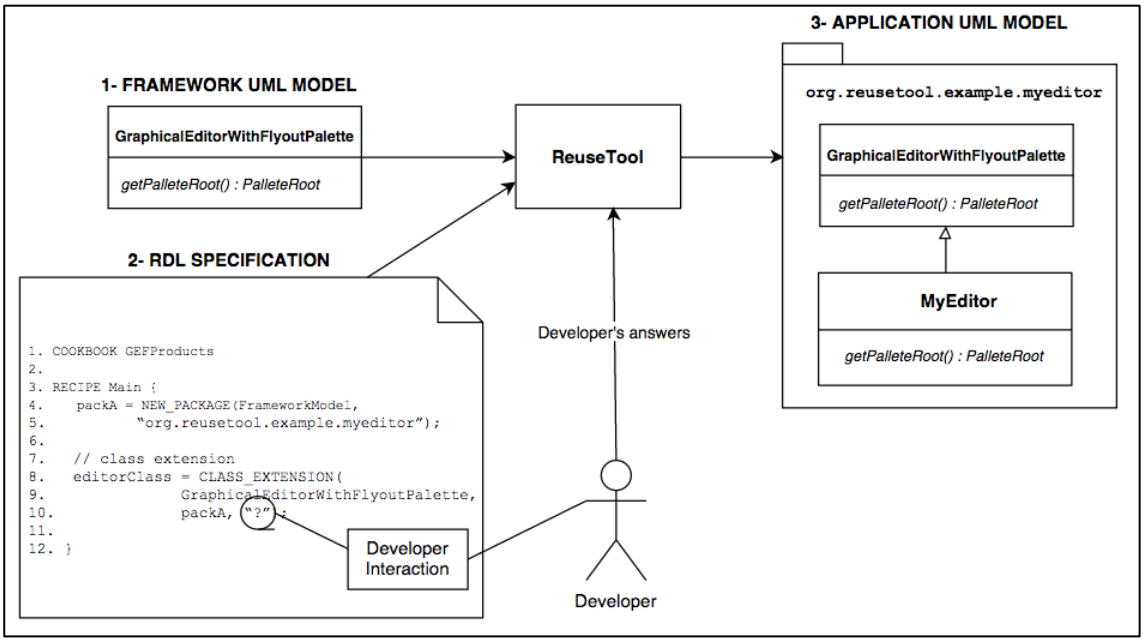


Figure 4. ReuseTool overview

Figure 4 depicts the elements involved in a framework instantiation using ReuseTool. The tool needs two inputs: i) the framework UML model; and ii) an RDL specification. RDL is a procedural language that offers commands to manipulate UML elements, and commands to indicate the flow of the instantiation process. Reuse Tool execution applies the changes defined in the RDL script to the framework UML model and generates the application UML model. ReuseTool interacts with the developer throughout the instantiation to collect the information needed to generate the final application model, such as the name of subclasses and to define if certain tasks should

be performed or not (e.g.: to determine if optional framework classes should be sub-classed in the framework application).

The structure of an RDL script is based on the structure of workflow processes, organizing human-tasks, system-tasks and control-flow structures. For instance, consider again the GEF Framework (GEF, 2002). Figure 5 illustrates the result of GEF instantiation to generate the Shapes application, one of the example application presented in Chapter 1. In Figure 5.1, an empty editor is created. Next, a tool palette is included in the editor (Figure 5.2). Figure 5.3 and Figure 5.4 show the addition of commands to the palette. Each of these steps involves the implementation of a hotspot. The creation of the whole application consisted, then, in the selection of hotspots and their implementation following a certain order.

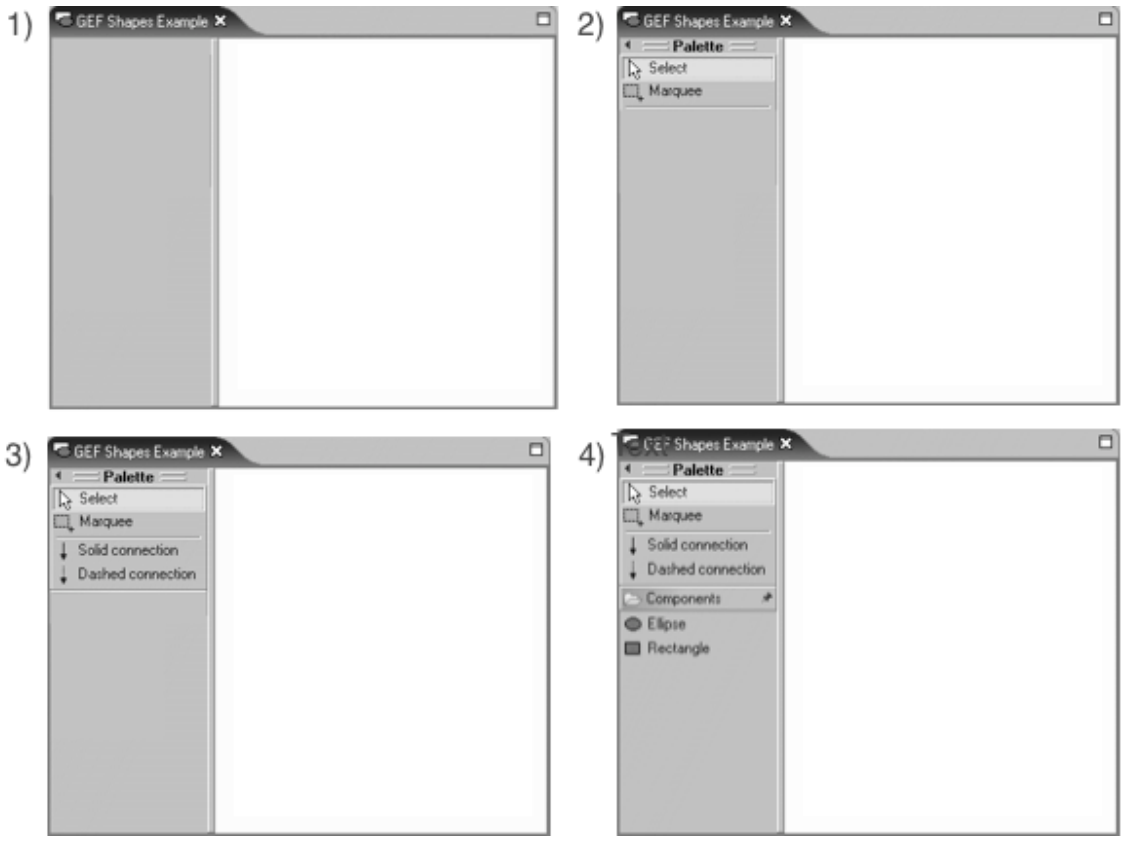


Figure 5. GEF Instantiation

Code 1 shows the RDL specification for the application in Figure 5. An RDL reuse process is represented using a script in the form of a cookbook. From Code 1, it is possible to highlight important RDL commands that denote reuse tasks, the *class extension* and the *method extension*. Examples of these commands can be observed in Code 1 Line 7 (class extension - `CLASS_EXTENSION`) and Code 1 line 10 (method extension – `METHOD_EXTENSION`). Another aspect of the RDL syntax that can be observed in these lines is the *interaction marker*, indicated by a question marker. This marker is used in the RDL script every time an interaction with the developer is required. For example, on line 7, the developer must inform the name of the application class while for methods extension this is not necessary and there is no need of user interaction.

Code 1 RDL Script example manually created based on the first two steps of a GEF tutorial[1].

```
1.  cookbook GEFProducts
2.
3.  recipe Main {
4.      packA = new_package(FrameworkModel, "org.reusetool.example.myeditor");
5.
6.      // class extension
7.      editorClass = class_extension(GraphicalEditorWithFlyoutPalette, packA, "?");
8.
9.      // method extension
10.     method_extension(GraphicalEditorWithFlyoutPalette,editorClass, getPaletteRoot);
11.     method_extension(GraphicalEditorWithFlyoutPalette, editorClass, doSave);
12.
13.     // class extension
14.     commandClass = class_extension(Command, packA, "?");
15.     method_extension(Command, commandClass, execute);
16.
17.     loop("Create new AbstractEditPart?") {
18.         editPartClass = CLASS_EXTENSION(AbstractGraphicalEditPart, packA, "?");
```

[1] GEF Tutorial Repository: Creating graphical editors with Eclipse and GEF. Available in: <https://github.com/vainolo/JJTV5_gef>. Accessed in 28 September 2014.

```
19.       method_extension(AbstractGraphicalEditPart, editPartClass, createFigure);
20.       method_extension (AbstractGraphicalEditPart,
21.       editPartClass, createEditPolicies);
22.       method_extension (AbstractGraphicalEditPart,
23.                 editPartClass, getModelChildren);
24.       method_extension (AbstractGraphicalEditPart,
25.                 editPartClass, update);
26.    }
27.
28.    interfaceClass = new_class(packA, "?");
29.    new_interface_realization(interfaceClass, packA, "?");
30.
31.    // class extension
32.    if ("Create XYLayoutEditPolicy?") {
33.       policyClass = class_extension(XYLayoutEditPolicy, packA, "?");
34.       method_extension (XYLayoutEditPolicy, policyClass, getCreateCommand);
35.    }
36. }
```

Besides commands for declaring the implementation of hotspots, RDL has also commands to indicate the control-flow of the instantiation process. This control-flow may include repetition of tasks, indicated by the `LOOP` command (Code 1 line 17), and conditionals, indicated by the `if-then-else` command (Code 1 line 32). Commands `cookbook` and `recipe` represent a program and a procedure, respectively. Every cookbook must have a recipe called Main that indicates the entry point of the instantiation process execution. The execution of activities in the RDL process follow the order in which the activities appear in the process specification (OLIVEIRA *et al.*, 2007).

The execution of RDL scripts by ReuseTool generates a UML Model for the framework application. Figure 6 illustrates a RDL specification and the Application UML Model generated by the execution of this script.

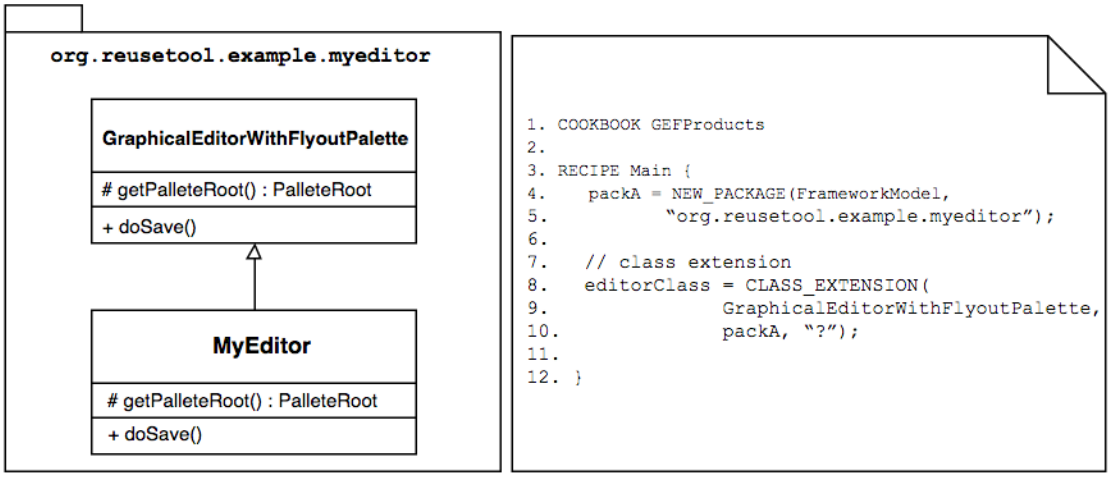


Figure 6. Application UML Model

2.3 Process Mining

Business Process Management (BPM) is a discipline that assembles the knowledge from information technology and management sciences and applies this knowledge to operational business processes. Under the umbrella of information systems used to support business process, there is a class of systems called Process-Aware Information systems (PAIS). As the name suggests, PAIS are systems that are aware of the underlying operational process they manage and execute. Examples of PAISs include larger ERP (Enterprise Resource Planning) systems (e.g. SAP and Oracle), CRM (Customer Relationship Management) systems, workflow management systems and call center software (AALST, 2013).

Process models play an important role in BPM as they can be used to assist different activities: i) provide insights on the process; ii) help structure discussion between stakeholders; iii) help finding inconsistencies; iv) assist performance analysis, v) serve as documentation and specification; vi) provide the information to configure the process to be executed (AALST, 2011).

According to its purpose, process models can be classified in formal models or informal models. Informal models are high-level models used for documentation and discussion. On the other hand, formal models are models used for analysis and the real execution of the process. In terms of representation, the formality of models can vary in a spectrum that goes from “Power point models” (informal models) to models represented by executable code (formal models). Informal models are more easily understood; however, they can be ambiguous and vague. Formal models may be too detailed to be understandable by stakeholders.

In spite of its type, a process model must be capable of correctly representing reality. Otherwise, they will only represent an idealized view of it and won’t be able to support the decision making process. In this context, the objective of process mining techniques is to mine process models that correspond to what happen in reality. To achieve this goal, process mining deal with data generated during the execution of real processes.

One of the main types of process mining is *process discovery*. It corresponds to the discovery of a process model that represents the business process of interest. Process discovery algorithms take an event log as input and produce a process model representing the dependencies between activities and the order in which they happened. The α-algorithm, the heuristic miner and the genetic miner are examples of algorithms for process discovery (AALST, 2011).

Next sections expose the main elements related to process discovery. Section 2.3.1 exposes how event logs are structured. In Section 2.3.2, a brief overview of process modeling notations is given. Finally, Section 2.3.3 discusses process discovery algorithms in more detail.

2.3.1 Event Logs

PAISs produce a vast amount of data. This data may be unstructured and scattered over many data sources. In addition, each information system may have its own internal data structure and its own way of representing this data. Therefore, a standardized way of representing process data so that it can be a suitable input for process mining algorithms is needed. In this scenario, a meta-model for process mining was proposed in (DONGEN, 2005). Figure 7 shows this meta-model. According to it, event logs must respect the requirements 1-4 listed below.

1. An event log usually corresponds to a unique process and contains a set of process instances, also called cases.
2. Each process instance corresponds to a single execution of the target process and contains an ordered set of events.
3. A sole event corresponds to the execution of one process activity. An event contains as many attributes as needed to describe the execution of the related process activity. Common attributes are the person who executed the activity, activity costs and timing information.
4. All events must be related to a unique process instance.

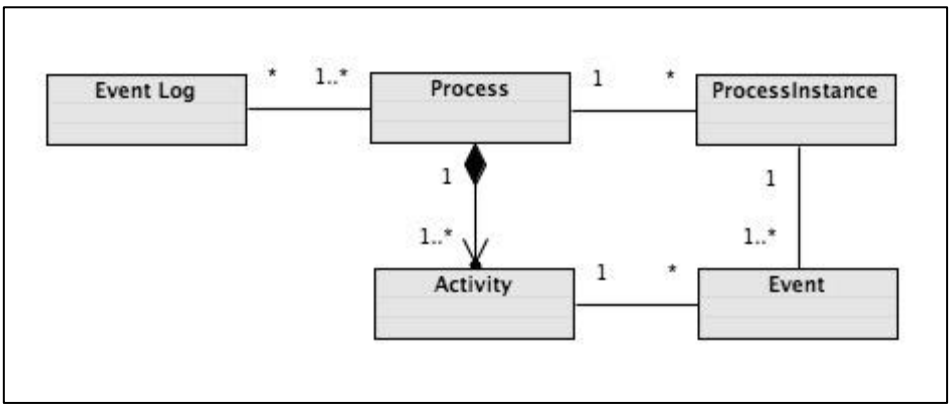


Figure 7. Process mining meta-model. Adapted from (DONGEN, 2005).

Table 1 shows an excerpt from an event log adapted from (AALST, 2011). The table contains information related to two process instances – cases id 1 and 2, respectively. The events in this process have three attributes: a timestamp, an activity and a resource – the person responsible for executing the activity. The timestamps indicate that these events are in an ordered sequence.

Table 1. An excerpt from an event log – adapted from (AALST, 2011).

CASE ID	EVENT ID	PROPERTIES		
		TIMESTAMP	ACTIVITY	RESOURCE
1	35654423	30/12/2010.11:02	Activity A	Pete
	35654424	31/12/2010.10:06	Activity B	Sue
	35654425	05/01/2011:15.12	Activity D	Mike
	35654426	06/01/2011:11.18	Activity E	Sara
	35654427	07/01/2011.14.24	Activity H	Pete
2	35654483	30/12/2010:11.32	Activity A	Mike
	35654485	30/12/2010:12.12	Activity C	Mike
	35654487	30/12/2010:14.16	Activity B	Pete
	35654488	05/01/2011:11.22	Activity D	Sara
	35654489	08/01/2011:12.05	Activity E	Ellen
	…			

According to the meta-model definition, it is possible to define a mapping from the data in information systems following its requirements. This mapping needs a standard data format. The first standard format used was the XML-based format called MXML (DONGEN, 2005). Later, a new format was proposed to overcome some of the limitations of MXML, the XES (eXtensible Event Stream) format (VERBEEK *et al.*, 2011). In 2010, the IEEE Task Force on Process Mining adopted this new format as

the current standard to represent event logs (AALST *et al*., 2012). Code 2 shows a fragment of the event log in Table 1 using XES.

XES extensibility is its main advantage over MXML. XES extensions provide proper semantic to the attributes in the event log by defining a set of attributes and information on how to interpret these attributes (VERBEEK *et al.*, 2011). For example, the event log in Code 1 uses the standard extensions: concept, time and organizational. The time extension is used to define how to record a timestamp in the event log and is indicated in Code 1 line four by `time:timestamp`. The organizational extension is used in the representation of an employee – `org:resource` –, but this extension is not limited to this. It can also be used to indicate roles (`org:role`) and groups (`org:group`).

In order to obtain a good outcome from the mining activity, the event log is expected to contain a representative sample of the process behavior. It means that the log should be *complete* and free of *noise*. Noise can be a result of wrong behavior, but also rare or infrequent behavior. The notion of completeness ensures that the log does not contain too little data for the mining task, i.e., it ensures that the log has sufficient sequences to represent the process behavior (AALST, 2011).

2.3.2 Process Models

A plethora of process modeling languages can be used to represent the discovered models, such as petri nets, workflow nets, Business Process Modeling Notation (BPMN) and causal nets (AALST, 2011). This Section gives a brief overview of each of these modeling notations.

Code 2. XES event log

```
<trace>
    <string key="concept:name" value="1"/>
    <event>
        <string key="concept:name" value="Activity A"/>
        <string key="org:resource" value="Pete"/>
        <date key="time:timestamp" value="2010-12-30T11:02:00.000+01:00"/>
        <string key="Activity" value="Activity A"/>
        <string key="Resource" value="Pete"/>
    </event>
    <event>
        <string key="concept:name" value="Activity B"/>
        <string key="org:resource" value="Sue"/>
        <date key="time:timestamp" value="2010-12-31T10:06:00.000+01:00"/>
        <string key="Activity" value="Activity B"/>
        <string key="Resource" value="Sue"/>
    </event>
    <event>
        <string key="concept:name" value="Activity D"/>
        <string key="org:resource" value="Mike"/>
        <date key="time:timestamp" value="2011-01-05T15:12:00.000+01:00"/>
        <string key="Activity" value="Activity D"/>
        <string key="Resource" value="Mike"/>
    </event>
    <event>
        <string key="concept:name" value="Activity E"/>
        <string key="org:resource" value="Sara"/>
        <date key="time:timestamp" value="2011-01-06T11:18:00.000+01:00"/>
        <string key="Activity" value="Activity E"/>
        <string key="Resource" value="Sara"/>
    </event>
    <event>
        <string key="concept:name" value="Activity H"/>
        <string key="org:resource" value="Pete"/>
        <date key="time:timestamp" value="2011-01-07T14:24:00.000+01:00"/>
        <string key="Activity" value="Activity H"/>
        <string key="Resource" value="Pete"/>
    </event>
</trace>

```

Petri-nets are the oldest and most investigated process modeling notation. It is a bipartite graph with two types of nodes, transitions and places. The graph has a static structure, where the firing of tokens determines the process flow. The distribution of tokens in the net indicates its current state, and it is called marking. At the initial stage, only the start node has a token.

Workflow-nets are a special type of petri-nets with a dedicated source place, where the process starts, and a dedicated sink place, where the process ends. All sequences are necessarily in a path from source to sink. Figure 8 illustrates a workflow-net for a process that handles a request for repair. In this representation, circles represent places and rectangles represent transitions.

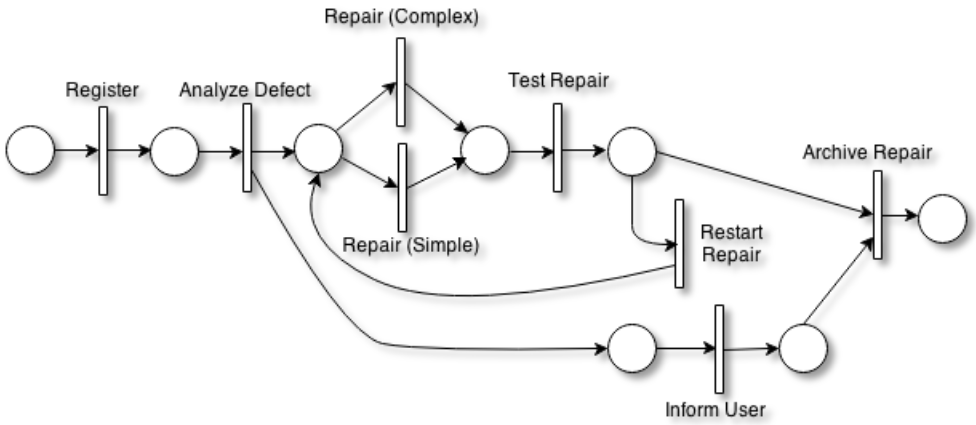


Figure 8. Workflow-net example[2].

BPMN (Business process modeling notation) is a graphical notation with activities and gateways. Gateways semantic give the routing logic, including control decisions, forking, merging and joining of paths. There are specific split and join gateways for AND, OR and XOR. In addition to tasks and gateways, BPMN also has event elements, which may impact on the flow of the process. Examples of events are the start event (initial process node) and the final event (last process node) (OMG, 2011). Figure 9 illustrates the Request for Repair process modeled in BPMN. Rectangles are activities, diamonds represent gateways and the circles are the start event and the end event (double-lined circle).

[2] Process modeled from the repair event log. Available in: <http://www.processmining.org/logs/start>. Accessed on 31 May 2015.

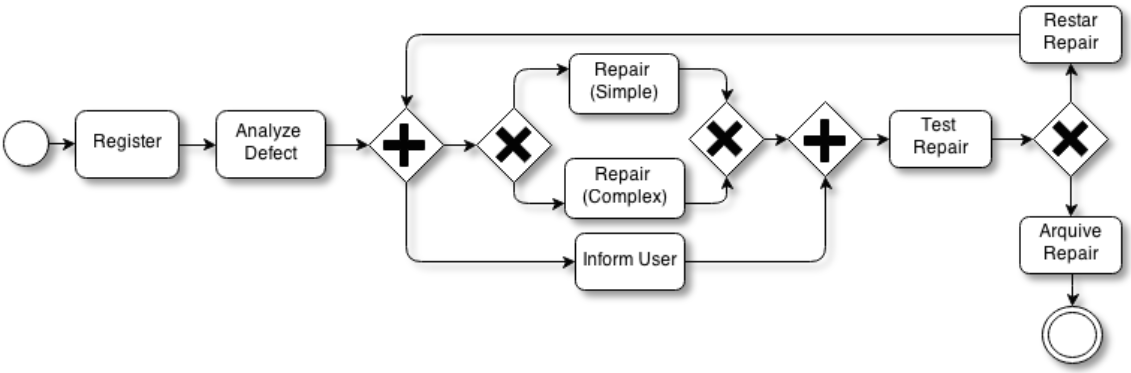


Figure 9. BPMN example2

Causal-nets are a notation designed for process mining. This notation eliminates the need for intermediate elements for routing purposes. It is also a graph-like notation with nodes and arcs, where node represent process activities and arcs represent causal dependencies between these activities. Each node has a set of possible input bindings and a set of possible output bindings that specify the process flow logic. Figure 10 shows a causal-net for the same process presented in Figure 8.

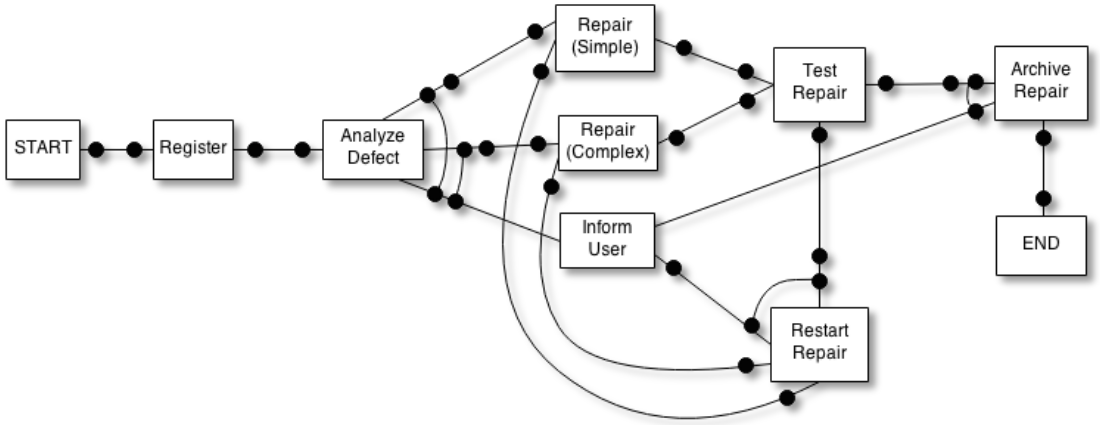


Figure 10. Causal-net example[2].

Input and output bindings are represented in a causal-net by the small black dot in the edges. Arcs connecting black dots indicate a set of activities as input or output binding. For example, the activity “Analyze defect” can be followed by the activity

“Repair (Simple)” alone, the activity “Repair (Complex)” alone or by those activities together with the activity “Inform User” ({“Repair (Simple)”,“Inform User”} or {“Repair (Complex)”, “Inform User”}) (AALST *et al.*, 2011).

2.3.3 Process Discovery Algorithms

The goal of process discovery algorithms is to find the simplest underlying model capable of describing the behavior of the data present in the event log. Different perspectives of the same process can be investigated, such as the control-flow perspective, the organizational perspective, the case perspective and the time perspective.

The control-flow perspective considers the ordering and causal dependencies between activities within a process to find a process capable of representing all possible paths in the event log. The organizational perspective is correlated to the actors in the process and how they relate to each other. The case perspective focuses on the properties of cases, which can also include the attributes related to the control-flow and the organizational perspectives. The time perspective investigates timing and frequency of events (AALST, 2011). In this work, we focus on the control-flow perspective.

The control-flow perspective aims at discovering causal dependencies between activities in a process. To this end, process discovery algorithms should be capable of correct mining the common control-flow constructs present in process models. These constructs are: sequences, parallelism, choices, loops, non-free-choice, invisible tasks and duplicate tasks (MEDEIROS, 2006). Table 2 gives the definition for each of these constructs. It is worth noting that not all process discovery algorithms are capable of mining all constructs in Table 2. For instance, the process modeling notation used by the algorithm can pose obstacles to the discovery of certain workflow constructs. That is called representational bias. If the chosen representation does not allow the process

to have two activities with the same label, for example, it is not possible to model duplicate tasks using this representation.

There are a variety of process discovery algorithms in the literature. Among the first methods proposed for process discovery from event data are the methods proposed in (COOK and WOLF, 1998). They proposed three methods for process discovery in the context of software processes, RNet (statistical approach), KTail (algorithmic approach) and Markov methods (statistical and algorithmic approach). In the context of workflow management, the first approaches were introduced by (DATTA, 1998) and (AGRAWAL *et al.*, 1998). Since then, many other algorithms have been proposed as listed in (WEERDT *et al.*, 2012). Many of these algorithms were implemented as a plugin for the ProM framework, a pluggable environment for process mining (DONGEN, 2005).

Table 2. Workflow constructs to be mined by process discovery algorithms (MEDEIROS, 2006).

WORKFLOW CONSTRUCTS	**DEFINITION**
Sequences	Indicate that activities must be performed in a pre-defined order, one after the other.
Parallelism	Indicate that the execution of one or more activities can be performed concurrently or independently.
Choices	Indicate situations where there is a choice between activities and only the selected activity will be executed.
Loops	Indicate the repetition of one of more activities.
Non-free-choice	Indicate a mix between choice and synchronization. In this case, choices occur when certain process constrains are met.
Invisible tasks	Silent steps used for routing purposes. Invisible tasks are not present in event logs.
Duplicate tasks	Existence of various tasks for a single label within a process.

Next sub-sections give an overview of two process discovery techniques: the α-algorithm and the heuristic miner. The α-algorithm is considered to be a good algorithm

to introduce the concepts behind process discovery techniques. Many of the underlying concepts used in this algorithm were applied to more complex and robust techniques. The heuristic miner was the chosen algorithm for the solution proposed in this dissertation. It mines a causal-net from the data in the event log. One of its main advantages is the ability to deal with noisy behavior (DONGEN *et al.*, 2009).

2.3.3.1 α-algorithm

The aim of this algorithm is to find a workflow-net that represents the behavior in the event log. For this task, it searches for control-flow patterns in the event log data. The identification of control-flow patterns is based on log-based ordering for the activities in the event log. A log-based ordering is related to the number of direct succession between tasks. For instance, if one activity A is always followed by an activity B, this may indicate a causal dependency between these activities. Table 3 summarizes the log-based relations used by the α-algorithm.

Table 3. Log-based ordering relations

RELATION	**NOTATION**	**DEFINITION**
Directly follows	$a >_L b$	iif for a given trace $\sigma = \langle t_1, t_2, t_3, \dots t_n \rangle$ and $i \in \{1, \dots, n-1\}$, such that $\sigma \in L$ and $t_i = a$ and $t_{i+1} = b$
Causality relation	$a \rightarrow_L b$	iif $a >_L b$ and $b \not>_L a$
Choice	$a \#_L b$	iif $a \not>_L b$ and $b \not>_L a$
Concurrent behavior	$a \|\|_L b$	iif $a >_L b$ and $b >_L a$

A footprint matrix stores the information about the log-based relations. A footprint matrix F is a matrix $n \times n$, where n is the number of activities in the process. Each item F_{ij} in this matrix indicates a relation between two activities (activity in line i and activity in column j), where the relation is one of the four from Table 3. Based on

the relations in the footprint matrix, it is possible to extract advanced ordering relations that lead to the workflow patterns in Table 4. Figure 11 gives a graphical representation of these patterns.

The event log in Table 1 can be used to illustrate the discovery of workflow patterns. If we consider only cases 1 and 2, it is possible to verify that activity *a* can be directly followed by activities *b* and *c*. Also, activity *a* is always the first activity so it does not follow *b* nor *c*. Therefore, there is a choice after the execution of activity *a* between tasks *b* and *c*. These dependencies correspond to a XOR-split construct, as indicated in Table 4 line 2.

Table 4. Workflow patterns (AALST, 2011)

WORKFLOW PATTERN	**RELATION**
Sequence	$a \rightarrow b$
XOR-split	$a \rightarrow b, a \rightarrow c$ and $b \# c$
XOR-join	$a \rightarrow c, b \rightarrow c$ and $a \# b$
AND-split	$a \rightarrow b, a \rightarrow c$ and $b \mid\mid c$
AND-join	$a \rightarrow c, b \rightarrow c$ and $a \mid\mid b$

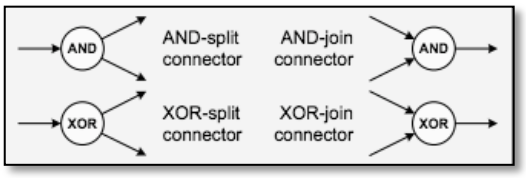


Figure 11. A graphical representation for workflow patterns in Table 4

It is worth noting that multiple workflows could fit the behavior in the log. However, the α-algorithm mines the "simplest" workflow fitting the log, which means that some constructs, such as AND-split and AND-join, may be omitted if not explicitly found in the log.

This algorithm assumes the event log to be complete with respect to the possible sequences in the process at hand. Also, it considers that there is no noisy behavior. As a consequence, the outcome is very sensitive to incompleteness and noise. Other limitations of this algorithm include its inability of discovering short loops and non-local, non-free choice constructs (WEERDT *et al*. 2012). Despite its limitation, the α-algorithm is able to mine useful process models provided that the underlying process does not require duplicated activities or silent transitions (transitions not recorded in the event log) (AALST, 2011).

There are other algorithms in the literature with an approach similar to the α-algorithm, the α-series algorithms. These algorithms were proposed to overcome some of the limitations of the α-algorithm. We refer to (DONGEN, 2009) for a better comparison between the α-series algorithms.

2.3.3.2 Heuristic Miner

The heuristic miner is considered to be a robust algorithm due to its capacity to deal with noisy behavior (WEIJTERS and MEDEIROS, 2006). This ability comes from the use of frequency information. Unlike the α-algorithm, causal dependencies in the event log are present in the final process models if and only if the number of occurrences is over some threshold.

In the first step of this algorithm, a dependency graph is created. Similarly to what occurs in the α-algorithm, it is possible to create a matrix D with the dependency values, where each matrix item D_{ij} stores the dependency value between the pair of activities in a row i and column j. Equation 1 gives a way of measuring the dependency for any pair of activities a and b ($a \Rightarrow b$) based on the directly follows relation ($>_L$), shown in Table 3.

$$|a \Rightarrow b| = \begin{cases} \dfrac{|a >_L b| - |b >_L a|}{|a >_L b| + |b >_L a| + 1} & if\ a \neq b \\ \dfrac{|a >_L a|}{|a >_L a| + 1} & if\ a = b \end{cases}$$

Equation 1. Dependency measure (AALST, 2011)

From the dependency matrix, a dependency graph can be generated. The nodes in this graph correspond to process activities. There is an edge between a pair of activities if the calculated dependency between them is greater or equals the defined threshold. The dependency graph provides the backbone for the target process.

The aim of the heuristic miner is to find a causal net for the process in the event log. Thus, the input and output bindings must be included in the dependency graph. This information is obtained by the replay of the event log in the dependency graph. A threshold is also used to determine which bindings to remain in the final causal net. For instance, the occurrence of an activity a followed by an activity b appears four times in the event log while the occurrence of an activity a followed by an activity c happens ten times. If the threshold is lower or equal to four, the output binding set of activity a is [{b}, {c}]. However, if it were greater than four, the output binding set would not include b.

2.4 Conclusion

This chapter discussed the use of object-oriented framework to improve software reuse. A domain analysis defines what a framework should offer in terms of immutable resources and flexible resources – *hotspots*. Immutable resources form the basis for all applications built from this framework while hotspots are offered to include application-specific code. Good documentation is essential for the framework success, describing all framework hotspots and how these hotspots should be implemented. Another

approach is the use of a tool to semi-automate the instantiation process. One of such tools is ReuseTool.

ReuseTool knows the instantiation process from a specification in RDL. This specification contains the commands on how to modify the framework UML model into the application model. Currently, the framework developer or a specialist should write the RDL specification due to their knowledge about the framework hotspots. In this dissertation, we propose the use of knowledge embedded in framework applications in order to mine this specification. This is performed with the use of process discovery algorithms, also discussed in this chapter along with an introduction to the process mining discipline. Next chapter, Chapter 3, will continue the discussion about framework reuse, presenting an informal review of related work that also propose the use of data-mining techniques solutions to improve reuse. Next, Chapter 4 depicts Reuse Miner in details, explaining all steps involved in the solution proposed.

3 Related Work

This chapter gives an overview of solutions to improve the reuse of object-oriented frameworks. This review focuses in data-mining based solutions that gather knowledge on how to reuse a framework based on previous framework instantiations.

3.1 Introduction

A barrier to the effective reuse of object-oriented frameworks is the lack of learning resources to support developers when learning how to reuse it (MATTSSON 1996; KIRK *et al.,* 2007). Developers need to know what are the framework's hotpots that need customization to build a new application. In the case of white-box frameworks, this is equivalent to knowing what classes the application need to subclass and which methods should be overridden/extended. Thus, quality framework documentation is of utmost importance. Mattsson (1996) argues that a framework is not reused unless it has a documentation that eases the task of reusing it. Besides framework documentation, framework examples are also recognized as an important resource (MATTSSON, 1996). Similarly, developers can use past framework applications for learning purposes.

In this context, several solutions have been proposed in the literature aiming at complementing the framework documentation and supporting developers while instantiating a framework based on what has been done in past framework applications. Table 5 lists some of the work found in the literature, indicating the solution purpose and the approach used. The following sections discuss these solutions in more details.

Table 5 Uses of data mining in framework development

PURPOSE	*INPUT*	*OUTPUT*	*APPROACH*	*REFERENCES*
Improving Framework Documentation	Analysis pattern languages	Framework hotspots	• Domain analysis / reverse engineering	• (BRAGA and MASIERO 2004; BRAGA and MASIERO 2003)
	Application code	Mining subclassing directives to improve framework reuse	• Clustering algorithm and metrics	• (BRUCH *et al.*, 2010)
API Usage recommendations	Application code	Reuse Patterns	• Association rules	• (MICHAIL, 2001)
	Code Search Results	Filtered search results	• Sequential patterns	• (XIE & PEI, 2006; Zhong *et al.*, 2009)
	Application code	Code recommendation (API) - Intelligent code completion system	• Frequency • Association rule • Best Matching Neighbors algorithm (BRUCH *et al.*, 2009)	• (BRUCH *et al.*, 2009)
	Application code	Code recommendation – API sequential pattern	• Sequential patterns	• (SILVA JUNIOR *et al.*, 2012)
Code Snippets Recommendation	User query	Searching API usage examples in code repositories with sourcerer API search	• APIs usage similarity	• (Bajracharya *et al.,* 2010)

	Queries built on top of framework elements	Code examples in the search results	• Statistical Analysis	• (THUMMALAPENTA & XIE, 2008)
Framework Hotspots	Execution Traces	Supporting Framework Use via Automatically Extracted Concept-Implementation Templates	• Dynamic Analysis	• (HEYDARNOORI, CZARNECKI, and BARTOLOMEI 2009; HEYDARNOORI et al. 2012)
Process Mining in Software Engineering	Event data from process execution	Process model	• Process Discovery	• (COOK and WOLF 1995)
	Data from software repositories in the form of event logs	Process model	• Process Mining (Business)	• (PONCIN, SEREBRENIK, and BRAND 2011)

3.2 Framework Documentation

BRAGA and MASIERO (2004) propose a generic process to retrieve hotspots from analysis pattern languages. Analysis pattern languages are composed of patterns to solve problems during the system analysis. The proposed process to find hotspots first analyzes the pattern language graph and the information in sections “Following patterns”, “Related patterns” and “Context” to identify patterns paths, gathering mandatory and optional patterns. Next, each patterns is analyzed individually to find other hotspots. Finally, each hotspot is refined to include information about its design and implementation. The final list of hotspots can be used to guide the design and implementation of frameworks for the current domain. Also, it can be used to document the framework instantiation, which can be either manual or automated as showed in (BRAGA and MASIERO 2003).

BRUCH *et al.* (2010) presents an approach for mining subclassing directives from application code. Subclassing directives are part of the framework documentation used to indicate how to subclass framework classes. Examples of these directives are “subclasses [must/may/should] extend this method”. The choice of which modal to use (must, may, should) depends on the importance level of the target method. For instance, a directive that says that a method must be extended indicates that this method must be in the application code and also that it must call the method implementation in the superclass.

Four directives are presented in this work: i) method overriding directives; ii) method extension directives; iii) method calls directives; and iv) class extension scenario. For the first three directives, metrics to calculate the importance level of each are proposed. The importance level designates which modal to be used. The last

directive indicates which methods should be used together and for that matter a clustering algorithm is used.

The evaluation of the proposed solution indicates that it is capable of finding relevant directives that were neglected in the framework documentation, improving the framework documentation completeness. Also, the approach provides data on how the framework is actually being used, since it may be difficult to know a priori which hotspots framework classes should provide. Although the class extension scenario indicates classes that should be instantiated together, this solution is not process oriented. Its main advantage is the integration with the developer's IDE, which allows the user to obtain information about the framework instantiation while she implements framework features.

3.3 API Usage Recommendation

This category lists works that mine API usage patterns. CodeWeb (MICHAIL, 2001) is a tool developed to find reuse patterns from application code based using association rules. This technique is based on rules of the type if/then, indicating that if an application contains the antecedent application class it is likely to have the consequent. The tool output is a list of patterns where each pattern contains a list of related classes and methods with their corresponding usage frequencies.

Another solution that focuses on APIs frequently used together is MAPO (Mining API usages from Open source repositories) (Zhong et al. 2009). When searching for resources to learn how to use a framework API, developers may use code search engines, such as Google Code Search and Black Duck Open Hub Code Search. These engines usually return a long list of results for a given query. What MAPO does is to filter the most relevant results from this list. The filtered result has

sequences of APIs frequently used together, in the order they are used. This approach helps developers in learning the possible usages of an API.

The Eclipse IDE offers a code completion functionality to help developers during the programming task. By default, suggestions are given based on what has already been typed and what is available for the object at hand. Both (BRUCH *et al.*, 2009) and (SILVA JUNIOR *et al.,* 2012) suggest improvements for this functionality, applying data mining techniques to filter the default results and present to the developer only the most relevant suggestions to his/her context.

In (BRUCH *et al.*, 2009), three different types of intelligent code completion system (CCS) are proposed: a frequency-based CCS, an association rule based CCS and the best matching neighbors CCS. The frequency-based CCS rates methods recommendation by their frequency of appearance in application code. The association rule based CCS has a similar approach to CodeWeb, but on a variable level. The last approach uses a modification of the k-nearest-neighbor machine learning algorithm called the best matching neighbors (BMN). In the evaluation of the intelligent CCS, the frequency-based performed poorly, finding only half of the relevant method calls and making wrong suggestions. The other two approaches have good results, with similar results for small patterns, but with the BMN approach having better results for larger ones. The BMN approach is available as an Eclipse plugin called Code Recommenders.

The second approach that leverages the code completion functionality of the developer's IDE (SILVA JUNIOR *et al.,* 2012) recommends a sequence of API calls based on frequent sequential patterns found in application code. The solution was also implemented as an Eclipse plugin and the evaluation showed that 71,6% of the recommendations were relevant for the developers participating in the experiment.

3.4 Code Snippets

SpotWeb (THUMMALAPENTA & XIE 2008) proposes an approach for discovering framework hotspots based on simple statistical analysis. Similarly to MAPO, it uses a search code engine (Google Code Search) for gathering application code. SpotWeb constructs queries for the search code engine from framework elements (classes, methods, interfaces, packages and constants). With the search result, it calculates usage metrics for each element and, then, it identifies which ones are hotspots by ranking elements using the usage metrics values. Also, it defines the concept of coldspots, which are framework areas rarely used. For the framework user, SpotWeb recommends code examples. For example, if the developer wants to know how to reuse a framework method, SpotWeb returns code examples containing usages of this method along with examples for existing dependencies of the method of interest. Once again, discrepancies between the result of mined and documented demonstrate the value of mining information from application code.

Sourcerer API Search (SAS) (Bajracharya *et al.*, 2010) is a search interface for finding API usages examples in software repositories. The interface contains three main components: a tag-cloud section that lists popular words found in the result; a code-snippet section to display the code snippets found for the current search; and a section to filter the top APIs used in the code snippets found.

The focus of the works in this Section is to assist the reuse task by providing better search results. They leverage on code search engines and usage information retrieved from software repositories to provide more relevant results to the user.

3.5 Framework Instantiation

HEYDARNOORI *et al.* (2009) expose FUDA (Framework API Understanding through Dynamic Analysis), an approach to improve framework reuse based on the extraction

of concept templates from applications execution traces. The implementation of framework applications follows an instantiation process. This process can be broken into smaller processes that implement framework concepts. In (HEYDARNOORI *et al.*, 2009), framework-usage templates describing the steps to implement a framework concept are mined from execution traces. The execution traces register the interactions between the application and the framework during runtime. The templates are written in Java and represent an approximation of the required and sufficient steps for the concept implementation, including information about packages that should be imported, framework classes to subclass, interfaces to implement, methods to implement, objects to create and methods to call. The template also includes information about methods call nesting and ordering.

In the experiment undertaken in (HEYDARNOORI *et al.*, 2012), FUDA presented relatively high precision and recall from only two applications and execution scenarios. Also, the use of templates along with the framework documentation in the implementation of concepts reduced the time for completing the task when compared to the implementation with the documentation alone.

FUDA's rationale has some similarities with Reuse Miner regarding the information included in the discovered templates. However, the focus of FUDA is in a specific concept, while Reuse Miner targets a process description for a complete application. After some experiments, it was observed that FUDA only needs two applications to obtain useful results. However, it requires the application to be installed and running which may represent an obstacle extra burden to the approach. In this sense, Reuse Miner has the advantage of dealing with static code.

3.6 Process mining in Software Engineering

One of the first uses of the term process discovery is attributed to COOK and WOLF (1995) in the context of Software Engineering. They propose an approach to discover software process models that captured the behavior of an on-going process based on event data generated during its execution. Their objective was to obtain a formal process representation to the software process so that software methods that assumed the existence of a process model could be used.

Later, PONCIN *et al.* (2011) proposed FRASR (FRamework for Analyzing Software Repositories). FRASR is based on the ProM framework and it handles the process mining over a variety of software repositories, dealing with the needed challenges to connect data from these multiple sources (bug trackers, mailing archives, code repositories) into a single event log. This information is mapped to the event log meta-model presented in Chapter 2, Section 2.3.1. The advantage of using a process mining-based approach is that, once the log is ready, a variety of process mining techniques become available to the analysis of the process at hand. So, FRASR assembles data from the software development process into an event log so that it can be later analyzed by those techniques.

Reuse Miner belongs to this category. Similarly to FRASR, the first step is the creation of an event log to the process discovery task. The main difference is that FRASR deals with the software development process while the Reuse Miner focuses in the instantiation process of object-oriented framework. For this reason, the Reuse Miner only deals with code repositories, while FRASR needs to connect multiple repositories used during the development process. Both works take advantage of the available process discovery techniques, focusing in the data preparation to generate an event log and in the a posteriori analysis of the discovered process.

3.7 Conclusion

This chapter discussed some existing solutions in the literature that use data-mining algorithms to improve framework reuse. Several works focuses on ways to improve the reuse of framework and libraries APIs. Additionally, we found two related work that apply process discovery to software development. The first use of the term *process discovery* is attributed to Cook and Wolf (1995) in the context of software engineering, focusing in software processes. Later, PONCIN *et al.* (2011) proposed FRASR, a tool to collect information about a software process from different data sources and assemble this information into a single event log. The goal of FRASR is to make software data available to be analyzed by algorithms developed in the context of Process Mining discipline, which expects an event log as input. Reuse Miner also uses process discovery based on event logs, but it is centered on framework instantiation processes. Next chapter discusses this approach in more details.

4 Mining Framework Instantiation Processes

This chapter exposes the solution proposed in this dissertation. First, a solution overview is presented. Then, it is broken into three parts: the event log mining, the process mining and the transformation of the BPMN process mined in RDL.

4.1 Reuse Miner

Object-oriented frameworks assemble the knowledge and common solutions for problems of a certain domain The framework customization corresponds to a framework instantiation and it involves the selection and implementation of the hotspots relevant to the developer's application (MARKIEWICZ *et al.*, 2000). This selection of hotspots, however, can be a difficult task to newcomers. To reduce the burden of knowing all framework hotspots beforehand, the development of applications from a framework can be automated with the assistance of a tool, such as ReuseTool.

ReuseTool assists developers in customizing the framework's hotspots to create a new application. In order to do this, the tool uses a RDL script that represents the instantiation as an executable process. As a result of its execution, ReuseTool generates the application UML Model, tailored to the developers' needs.

Currently, ReuseTool depends on the framework developer or on a specialist to specify the framework instantiation process in RDL, which represents an extra burden on the framework developers and limits the ReuseTool adoption. An interesting alternative to developers would be to leverage on the knowledge embedded in framework instances to discover the reuse process. Based on this rationale, this chapter presents Reuse Miner, an approach created to mine RDL processes from code repositories that contain instances of a given object oriented framework.

Consider again the graphical framework GEF and Shape and Logic applications. These applications will be used throughout this Chapter to support the understating of the proposed solution. Figure 12 and Figure 13 depict some GEF hotspots implemented in Shapes and Logic applications, respectively. The diagrams represent the hotspots dependencies – a hotspot A depends on another hotspot B if hotspot A needs hotspot B for its implementation to be completed. Table 6 gives a brief definition for the hotspots in Shapes and Logic editors to help understand their role in the framework application.

Table 6. Some of GEF hotspots and their definition according to the framework documentation.

GEF HOTSPOTS	***DEFINITION***
GraphicalEditorWithFlyoutPalette	It is an implementation of a graphical editor that serves as a quick starting point for newcomers
EditPartFactory	A factory for creating Edit parts
EditPart	Represent the controller in the MVC pattern, connecting model and views.
AbstractGraphicalEditPart	Default implementation for Graphical Edit parts
AbstractConnectionEditPart	Base implementation for representing connections
EditPolicy	EditPolicies contribute to the editing behavior of an EditPart. Figure 12 and Figure 13 contain some available implementations, such as XYLayoutEditPolicy, ComponentEditPolicy and ConnectionEditPolicy

We can highlight some similarities between Shapes and Logic based on their diagrams. Both editors extend the *GraphicalEditorWithFlyoutPalette* to create their customized editor. These editors contain an implementation of an *EditPartFactory*, responsible for the creation of edit parts. With respect to the edit parts, the applications extend the framework classes *AbstractGraphicalEditParts*, for graphical elements, and *AbstractConnectionsEditPart*, for connections. A wider variety of edit policies is

depicted in the diagrams, defining how edit parts should be edited. According to this hotspot sample, we may manually create the RDL script in Code 3.

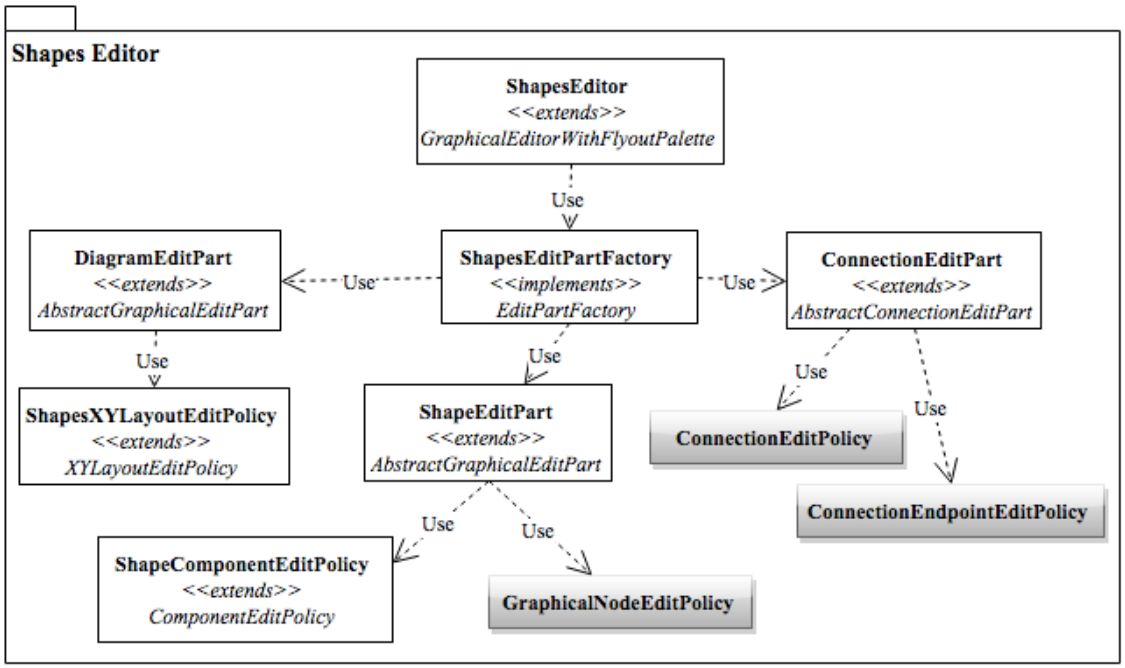


Figure 12. Implementation of some GEF hotspots in Shapes application

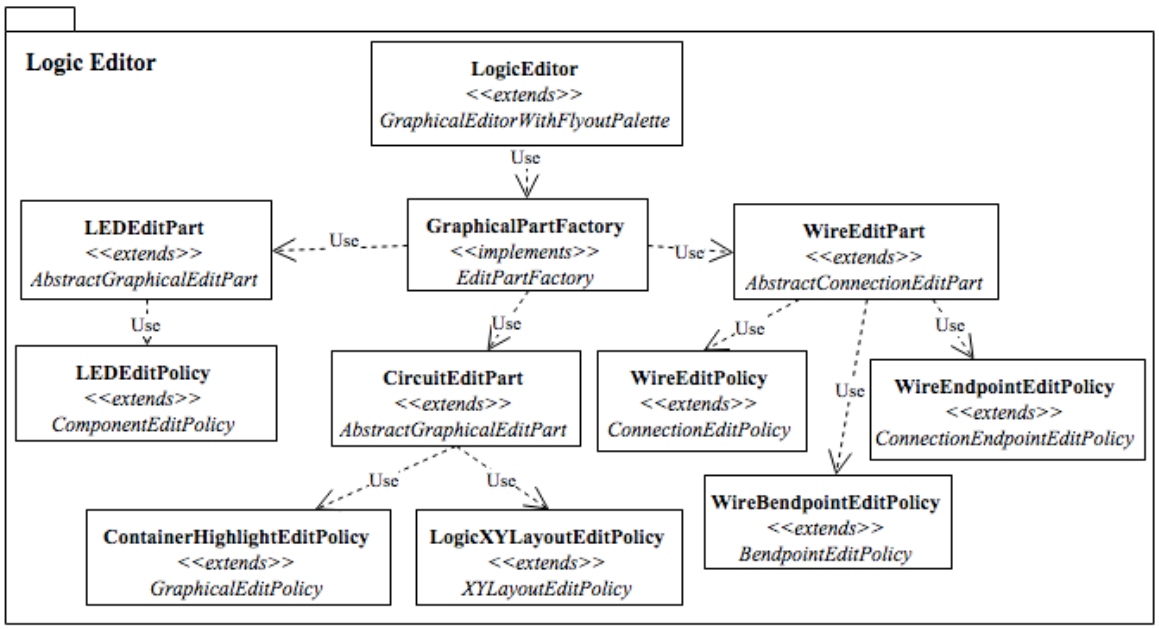


Figure 13. Implementation of some GEF hotspots in Logic application

Code 3. RDL for GEF framework manually created

```
1.  cookbook GEF
2.  recipe Main {
3.
4.     packA = new_package(FrameworkModel, "?");
5.     graphicalEditor = class_extension("org.eclipse.gef.ui.parts.
6.                          GraphicalEditorWithFlyoutPalette", "?");
7.     editPartFactory = new_class("?");
8.     new_interface_realization(editPartFactory,
9.                          "org.eclipse.gef.EditPartFactory");
10.
11.    loop("Create new edit part?") {
12.       editpart = class_extension("org.eclipse.gef.editparts.
13.                          AbstractGraphicalEditPart",
14.                          packA, "?");
15.
16.       if ("Create component edit policy?") {
17.          class_extension("org.eclipse.gef.policies.
18.                          ComponentEditPolicy", packA, "?");
19.       }
20.
21.       if ("Create graphical edit policy?") {
22.          class_extension("org.eclipse.gef.policies.
23.                          GraphicalEditPolicy", packA, "?");
24.       }
25.
26.       if ("Create XYLayout edit policy?") {
27.          class_extension("org.eclipse.gef.policies.
28.                          XYLayoutEditPolicy", packA, "?");
29.       }
30.    }
31.
32.    class_extension("org.eclipse.gef.editparts.
33.                AbstractConnectionEditPart", packA, "?");
34.
35.    loop("Create a new edit policy for connection") {
36.
37.       if ("Create connection edit policy?") {
38.          class_extension("org.eclipse.gef.policies.
39.                          ConnectionEditPolicy", packA, "?");
40.       }
41.
42.       if ("Create connection endpoint policy?") {
43.          class_extension("org.eclipse.gef.policies.
44.                       ConnectionEndpointEditPolicy", packA, "?");
45.       }
46.
47.       if ("Create bendpoint edit policy?") {
48.          class_extension("org.eclipse.gef.policies.
49.                       BendpointEditPolicy", packA, "?");
50.       }
51.    }
52. }
```

The description of the instantiation process using RDL contains two types of information: I) what are the hotspots involved in the instantiation (e.g.: class extension commands), and II) a valid flow to implement the hotspots in order to obtain a final application (sequence of actions, repetition of activities and conditionals). In Code 3, the commands to modify framework hotspots were obtained directly by identifying the framework classes and interfaces that were used by the example applications. To define the process flow, we made the following assumptions:

1. ***Conditionals*** were included in the RDL for classes that did not appear in both applications. This can be observed in the instantiation of edit policies in lines 16-29 and lines 37-50;
2. ***Loops*** (repetition) were used to represent hotspots that emerged more than once. In Code 3, they represent the creation of edit parts in lines 11 and 35;
3. The **sequence** of activities followed the dependencies levels. For example, for this script, the editor was considered to be the first implementation activity, followed by the creation of edit parts and, subsequently, the creation of edit policies.

The goal of Reuse Miner is to obtain RDL specifications such as the one in Code 3 from mining software repositories. To do so, the information about hotspots and the process flow must be extracted from application code. Figure 14 illustrates the proposed approach split in 3 steps. First, the *Log Mining* step (Figure 14.1) is responsible for the identification of hotspots from application code in software repositories. These hotspots are represented using event logs, the input for process discovery algorithms used in the subsequent step – *Process Mining* (Figure 14.2). Finally, the process discovered is translated to RDL to be reused in the development of new applications (Figure 14.3).

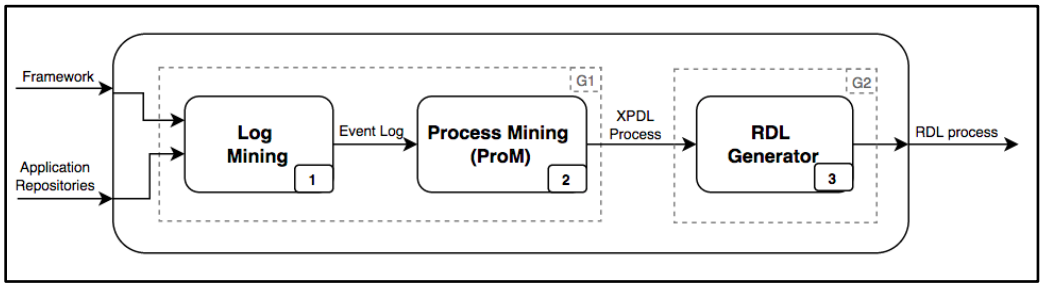


Figure 14. Solution Overview

The remaining of this chapter explains each step of this solution in more details. Section 4.2 exposes the Log mining step; Section 4.3 explains the process discovery step. Lastly, Section 4.4 deals with the translation of a discovered process to RDL.

4.2 Event Log Mining

The process mining algorithms developed in the context of business management rely on the existence of an event log with information about process executions, such as the activities performed and people responsible for their execution. However, each information system may use its own format when logging information regarding certain process. This motivated the development of the XES format, the current standard for the representation of event logs (VERBEEK *et al.*, 2011). The first phase of the proposed solution deals with the preparation of a XES event log for the target framework.

This step is of utmost importance to the solution since the quality of discovered process is directly related to the quality of the event logs used as input. Table 7 lists a few challenges that need to be tackled when extracting event logs (AALST, 2011). The next subsections will explain how the event log miner deals with each of these challenges. Scoping and Correlation are the topics of Section 4.2.1. Section 4.2.2 explains how granularity is taken into account when mining the framework instantiation process while Section 4.2.3 handles the ordering of events. Section 4.2.4 introduces

the *dependency tree* structure, data structure used to organize applications reuse actions considering all definition from Sections 4.2.1 to 4.2.3. Lastly, Section 4.2.5 gives details about the event log miner implementation.

Table 7. Challenges on event log extraction (AALST, 2011)

Scoping	*When dealing with a big amount of data, it is important to define and filter which data is relevant to the questions being studied.*
Correlation	Events in an event log are grouped by traces, which means that events in the same trace must be somehow related to each other. This is particularly challenging for event logs that use data from a variety of resources (e.g. data from multiple systems).
Granularity	Accounts for the desired level of detail of the events in the log. It may be necessary to abstract from low level events that are too detailed for the task as hand.
Ordering	Trace events need to be ordered so the extraction should try to reconstruct the ordering in which they happened

4.2.1 Scoping and Correlation

The extraction of event logs from existing data sources involves the identification and organization of events following the process mining meta-model introduced in Chapter 2, Section 2.3.1. According to this meta-model, illustrated in Figure 15a, an *Event* represents the execution of process activities (*Activity*) in a single process execution (*ProcessInstance*). This way, when recreating the event log for a process, it is necessary to correctly correlate events and their process instances. Moreover, events in a process instance must be ordered so that dependencies can be extracted using process discovery algorithms.

Reuse Miner proposes the mapping in Figure 15b to create events logs for the instantiation process of object-oriented frameworks. In this mapping, an *Event Log* stores information about a single process, the framework instantiation process,

indicated by *Framework process*. A *process* is composed by a set of activities; for the framework instantiation process, activities are framework reuse actions (*Reuse Actions*). Each process execution corresponds to a *Process Instance,* which maps to a *Framework Application*.

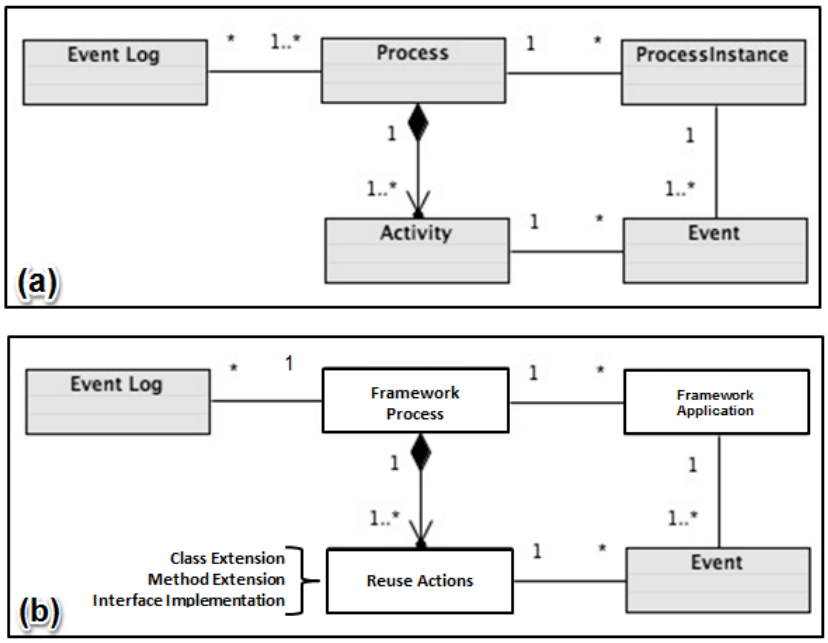


Figure 15. Log mapping

Once the mapping is defined, we need to plan the data extraction, considering the challenges in Table 7. Reuse Miner performs data extraction over software repositories that contain former applications of the framework for which we want to find the instantiation process. First, we need to define the scope of our data, defining which kind of information we need to search in the application repositories to support the discovery of the framework instantiation process. In Reuse Miner, three types of reuse actions are of interest: class extensions, interface implementations and method extensions. These types are the object-oriented actions that RDL is currently capable of representing. Code 4 shows a class extension example from a GEF application. In this example, we can observe a class extension (line 11) and also methods extension,

which are methods with the `@override` annotation. These are examples of the types of reuse actions that will be used in the event log.

All events in a trace must be correlated to a single framework application. Although Reuse Miner uses a variety of software repositories for data extraction, events in different repositories have no correlation whatsoever because each repository represents an unique *Framework Application*. This way we ensure that all events from an application repository are correlated and that they can be grouped in a single trace.

Code 4. Class extension example

```
1  package br.cos.ufrj.geftutorial;
2
3  import org.eclipse.core.runtime.IProgressMonitor;
4  import org.eclipse.gef.DefaultEditDomain;
5  import org.eclipse.gef.palette.PaletteRoot;
6  import org.eclipse.gef.ui.parts.GraphicalEditorWithFlyoutPalette;
7
8  import br.cos.ufrj.geftutorial.model.Canvas;
9  import br.cos.ufrj.geftutorial.parts.GEFEditorEditPartFactory;
10
11 public class GEFEditor extends GraphicalEditorWithFlyoutPalette {
12
13         public GEFEditor() {
14             setEditDomain(new DefaultEditDomain(this));
15         }
16
17         @Override                                   Superclass Method
18         protected PaletteRoot getPaletteRoot() {
19                 return new GEFEditorPalette();
20         }
21
22         @Override                                   Superclass Method
23         public void doSave(IProgressMonitor arg0) {
24         }
25
26         @Override                                   Superclass Method
27         protected void initializeGraphicalViewer() {
28                 super.initializeGraphicalViewer();
29                 getGraphicalViewer().setEditPartFactory(new GEFEditorEditPartFactory());
30                 getGraphicalViewer().setContents(new Canvas());
31         }
32 }
```

Framework class

Extends framework class

4.2.2 Granularity

When working with process mining, it is important to pre-process event logs to remove noisy behavior (infrequent activities) that may lead to spaghetti processes. While

working with a variety of repositories and a variety of applications, it is important to keep in mind that this variability might produce noise. To mitigate the impacts of infrequent reuse actions in the process discovery task, Reuse Miner proposes the mining of sub-processes, reducing the level of details of the main process.

An object-oriented framework exposes flexible points to developers so that they can create new applications taking advantage of the framework core implementation. A framework application can be delineated by the set of framework features it implements. Each individual feature can, in turn, be described by a set of steps, i.e., a process. For instance, consider the implementation of a GEF edit part. It involves the extension of an Edit Part class and the selection and implementation of the edit policies of interest, as shown in RDL in Code 3. Therefore, an edit part could have its own process, which describes its implementation details, and we could have a framework instantiation process that abstracts the implementation details of individual features. Code 5, Code 6 and Code 7 illustrate this rationale for the RDL process in Code 3. In Code 5 and Code 6, we have individual processes for edit parts (graphical edit parts and connection edit parts). While Code 7 presents a simplified process for the entire application, with references to the individual recipes that can be considered sub-processes of the main instantiation process. Recipes are called by their names as in Code 7 lines 12 and 14.

Extending the sub-process definition to features, we can also say that a framework feature is implemented by a set of other framework features, i.e., a framework feature can also be defined by a process that may contain features sub-processes. The event log miner uses this concept of sub-processes to reduce the granularity level of the process and to improve the results of the process discovery algorithm. A structure called *dependency tree* provides the information needed to find event logs for processes and sub-processes, as Section 4.2.4 explains in more details.

Code 5. RDL process for the creation of GEF edit parts.

```
1.  recipe EditPart {
2.
3.      // Repetition
4.      loop("Create new edit part?") {
5.          editpart = class_extension(
6.              "org.eclipse.gef.editparts.AbstractGraphicalEditPart", packA, "?");
7.
8.          // Conditional
9.          if ("Create component edit policy?") {
10.             class_extension(
11.                 "org.eclipse.gef.editpolicies.ComponentEditPolicy", packA, "?");
12.         }
13.
14.         // Conditional
15.         if ("Create graphical edit policy?") {
16.             class_extension(
17.                 "org.eclipse.gef.editpolicies.GraphicalEditPolicy", packA, "?");
18.         }
19.
20.         // Conditional
21.         if ("Create XY layout edit policy?") {
22.             class_extension(
23.                 "org.eclipse.gef.editpolicies.XYLayoutEditPolicy", packA, "?");
24.         }
25.     }
26. }
```

Code 6. RDL process for the implementation of a GEF connection edit part

```
1.  recipe ConnectionEditPart {
2.  connection = class_extension("org.eclipse.gef.editparts.
3.              AbstractConnectionEditPart", packA, "?");
4.
5.      // Repetition
6.      loop ("Create new edit policy for connection?") {
7.
8.          // Conditional
9.          if ("Create connection edit policy?") {
10.             class_extension("org.eclipse.gef.editpolicies.
11.                         ConnectionEditPolicy", packA, "?");
12.         }
13.
14.         // Conditional
15.         if ("Create connection endpoint edit policy?") {
16.             class_extension("org.eclipse.gef.editpolicies.
17.                         ConnectionEndpointEditPolicy", packA, "?");
```

```
18.          }
19.
20.          // Conditional
21.          if ("Create bendpoint edit policy?") {
22.             class_extension("org.eclipse.gef.editpolicies.
23.                          BendpointEditPolicy", packA, "?");
24.          }
25.       }
26.    }
```

Code 7. GEF process without the implementation details of edit parts

```
 1.   cookbook GEF
 2.   recipe Main() {
 3.      packA = new_package(FrameworkModel, "?");
 4.
 5.      graphicalEditor = class_extension("org.eclipse.gef.ui.parts.
 6.               GraphicalEditorWithFlyoutPalette", packA, "?");
 7.
 8.      editPartFactory = new_class("?")
 9.      new_interface_realization(editPartFactory,
10.                           org.eclipse.gef.EditPartFactory);
11.
12.      EditPart();
13.
14.      ConnectionEditPart();
15.   }
```

4.2.3 Ordering

Keeping a consistent ordering among cases is critical to the process discovery step. At the source code level, there is no ordering relationship between classes and methods. Nonetheless, classes hold dependencies between each other, as previously represented in Figure 12 and Figure 13. If we consider a hierarchy of hotspots dependencies, where a hotspot A is dependent on hotspot B when hotspot A depends on hotspot B for its implementation and both classes A and B extend a framework hotspot, we will be able to find similar structures among applications.

Along with class dependencies information, we will use commit information to help organizing these classes in chronological order. A repository knows the history of

the application development in the form of commits. Each commit has a list of modifications made to the repository, including all files added, modified and/or deleted, as in the example in Figure 16 which corresponds to the first commit made to a repository. To list reuse actions, we need to iterate over all files in the application commits history, identifying classes and methods that were implemented as an extension to a framework hotspot. The difference between a commit and its predecessor determines which reuse actions were implemented after the last changes committed to the repository. Reuse actions in subsequent commits are considered to be in a sequence. A *dependency tree* organizes all information about commit ordering and hotspots dependencies and it is the main structure used to generate event logs. Section 4.2.4 presents these trees.

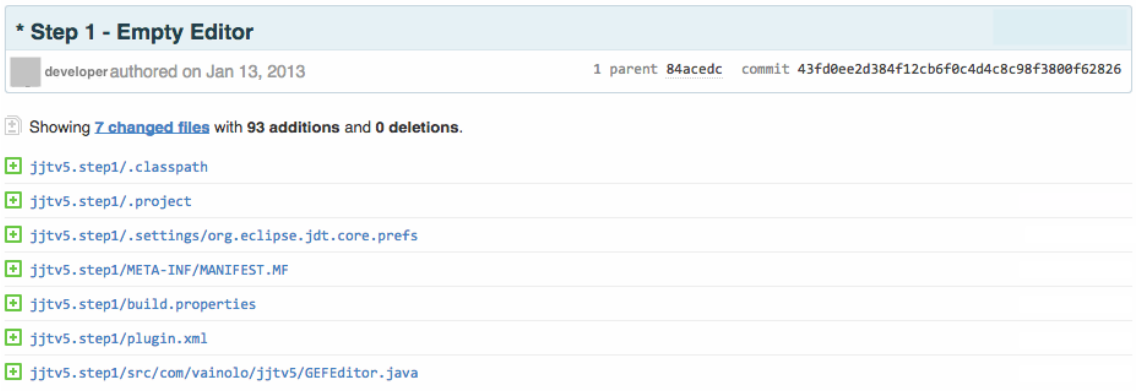


Figure 16. Commit example

4.2.4 Dependency Tree

A dependency tree is a tree structure used by Reuse Miner to organize the hotspots extracted from application code. A tree is created for each application. Tree nodes contain the following information: the hotspot name, the commit index and a list of child nodes that correspond to the tree node dependencies. An extra node is added to the dependency tree to accommodate nodes that are at the root level. These nodes are not

dependencies and they compose event log for the main process. Sub-processes event log are generated based on the other tree nodes.

Consider the dependency tree mined for a GEF tutorial application in Figure 17. It represents a small GEF application used as a tutorial for newcomers. It was chosen instead of Shapes and Logic due to its reduced size. This dependency tree contains all application classes that were included as an extension of a framework class. Each node indicates the class name, the superclass name (framework class name) and the commit index. The commit index indicates in which commit the application class was created – the oldest commit index is 1. The dependencies relationships give an initial indication of which are the classes involved in the implementation of a framework feature. For example, to implement the *GEFEditor*, the application implemented an EditPartFactory (*GEFEditPartFactory*) and a PaletteRoot (*GEFEditorPalette*). These dependencies can be identified in the GEFEditor class code, shown in Code 4. Classes without a parent with a dependency relationship are assembled under the tree root node.

Table 8 shows the list of traces that can be extracted from the dependency tree. Each trace contains a list with the names of the hotspots involved in the process of interest. The *root node* generates a trace that goes to the main event log, the event log that contains the information to delineate the flow of the instantiation process. Although the root node is not related to a hotspot implementation, it is included in the trace as a reference to the end of the process execution. For other tree nodes, the list of children also defines the trace with the node itself included at the end of the trace. For example, the GEFEditorPalette node contains two children: LinkFactory and NodeFactory, both implementing the framework interface CreationFactory. The trace for PaletteRoot is then CreationFactory – CreationFactory – PaletteRoot. Only nodes with children generate traces. Children nodes are sorted using the following criteria: I) sort children

by commit index; II) sort nodes that belong to the same commit by their hotspot name. III) for sub-processes, the last node in the trace always correspond to the sub-process of interest.

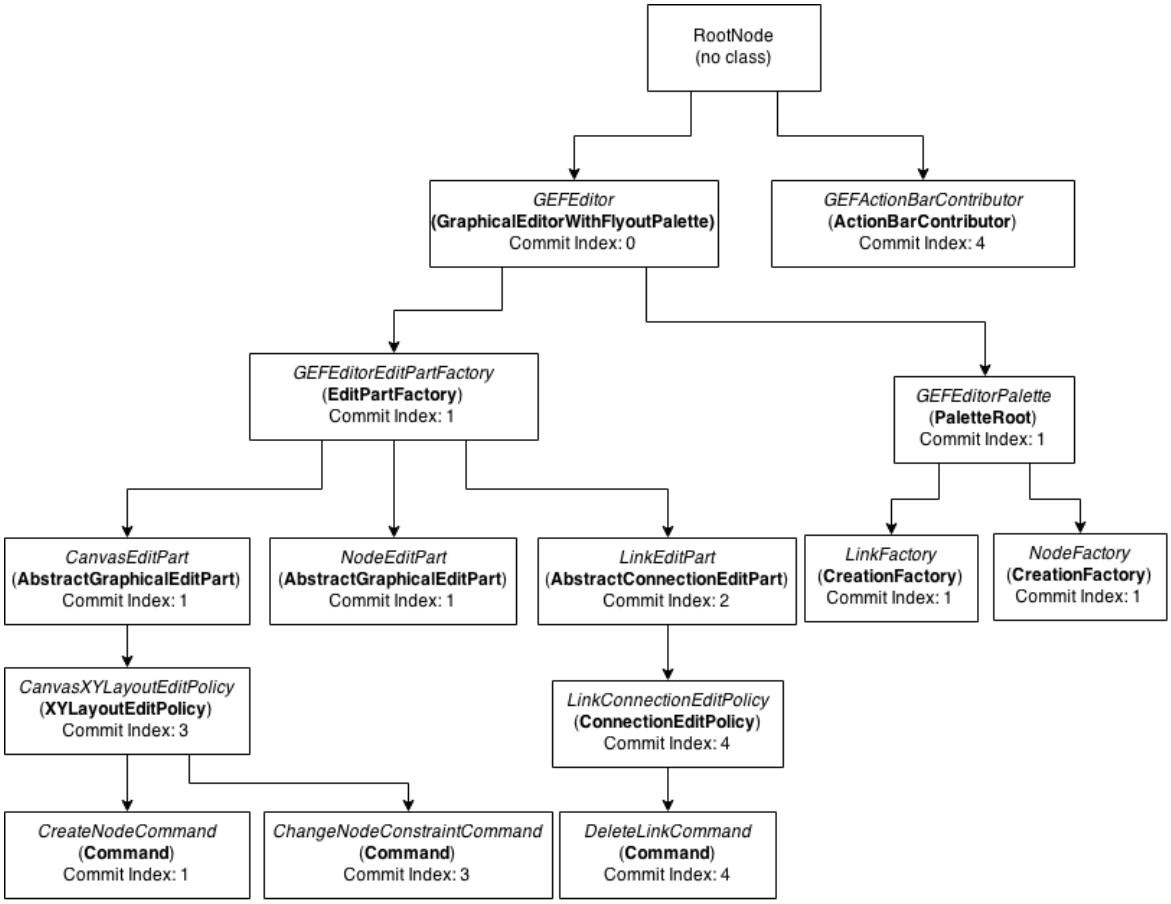


Figure 17. Dependency tree example

Table 8. Traces extracted from the dependency tree in Figure 17

PROCESS OF INTEREST	***TRACES***
Framework Application (root node)	GraphicalEditorWithFlyoutPalette – ActionBarContributor – root node
GraphicalEditor WithFlyoutPalette	EditPartFactory – PaletteRoot - GraphicalEditorWithFlyoutPalette
EditPartFactory	• Command - Command – XYLayoutEditPolicy – AbstractGraphicalEditPart – EditPartFactory • AbstractGraphicalEditPart – EditPartFactory • Command – ConnectionEditPolicy – AbstractConnectionEditPart –

	EditPartFactory
PaletteRoot	CreationFactory – CreationFactory - PaletteRoot

Each trace is included in the event log of the process of interest. For Table 8, we have traces for four distinct event logs (root event log, GraphicalEditorWithFlyoutPalette, EditPartFactory and PaletteRoot). Traces must be generated for all mined application and grouped in their respective event log. Then, these event logs can be used to discover processes in the ProM framework, as explained in Section 4.3. Code 8 presents the algorithm to create a dependency tree for a framework application.

Code 8. Dependency Tree Algorithm

```
1.  FrameworkApplication app = new FrameworkApplication(applicationRepositoryURL);
2.  DependencyTree tree = new DependencyTree();
3.
4.  for (Commit commit: app.getCommits()) {
5.      // Reuse actions and their dependencies are retrieved from the AST
6.      List<Event> commitReuseActions = commit.getReuseActions();
7.
8.      for (Event event : commitReuseActions) {
9.          // A node is added to the dependency tree, if it does not already exists
10.         TreeNode node = new TreeNode(event, commit);
11.         if (!tree.contains(node)) {
12.             tree.add(node);
13.         }
14.     }
15.
16.     // Next, we create tree nodes for the event dependencies
17.     for (EventDependency dependency : e.getDependencies()) {
18.         Treeode dependencyNode = new TreeNode(dependency.getEvent, commit);
19.
20.         if (!tree.contains(dependencyNode)) {
21.             // No node for the dependency event was found
22.             tree.add(dependencyNode);
23.         } else {
24.             // A node exists, but it already has a parent
25.             if (dependencyNode.parent != node) {
26.                 // New node is created
27.                 tree.add(dependencyNode)
28.             }
29.         }
30.         // If node and dependency node are not connected, add a new tree edge
31.         if (!tree.containsEdge(node, dependencyNode)) {
32.             tree.addEdge(node, dependencyNode);
33.         }
```

```
34.
35.         }
36.     }
37.
```

4.2.5 Log Miner Plugin

Figure 18 presents the UML model for the Event log miner. This UML model expresses the event log mapping proposed in Figure 15. The log miner was implemented as an Eclipse plugin and the model in Figure 18 was implemented with the assistance of EMF (Eclipse Modeling Framework), a modeling framework with tools that support Java code generation and adapter classes for viewing and editing the model. This plugin has four main functionalities: Parse Framework, Find and List Repositories, Mine Repositories and Generate XES log.

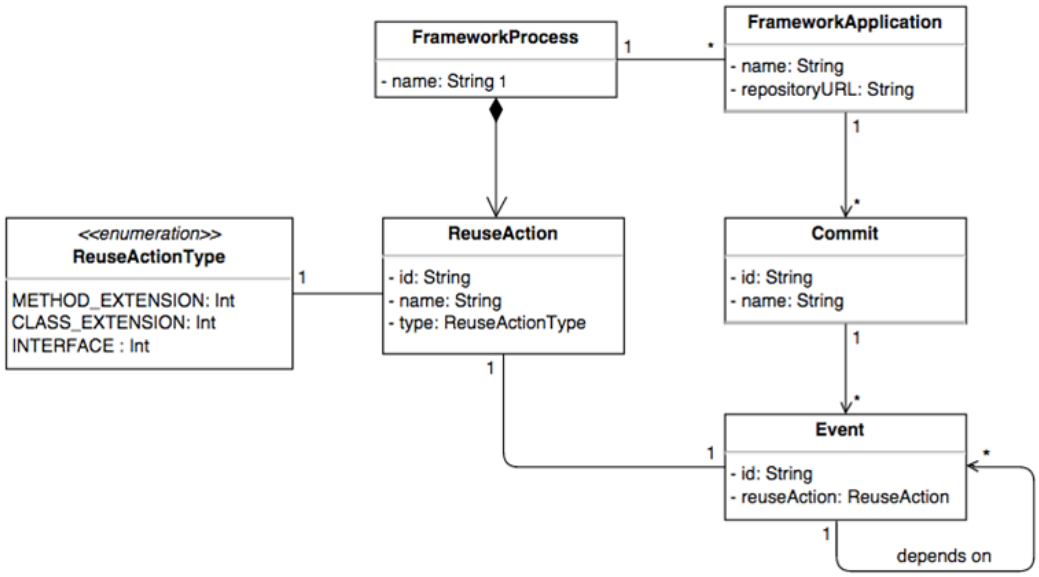


Figure 18. Event log miner UML model

1. Parse Framework: This action parses a framework, extracting classes, methods and interfaces. The current parser implementation targets only Java frameworks. An excerpt of GEF parsing is displayed in Figure 19, with the hotspots already mentioned in this Chapter.

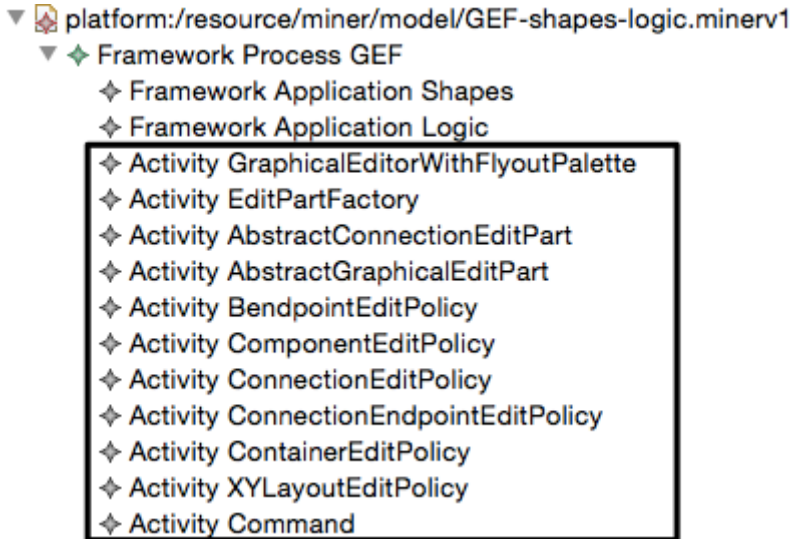


Figure 19. An excerpt of the result of GEF parsing

2. Find and List Repositories: Integration with the GitHub[3] API to assist the search for application repositories. It searches for a given keyword in GitHub repositories title, description and name. The repositories found are listed under the Framework Process node, such as the frameworks applications Shapes and Logic in Figure 19. The list is not final and should be reviewed to remove repositories not suitable for the mining task. It might include repositories that do not belong to the target framework and also non-representative framework applications.

3. Mine Repositories: This is the core functionality of the Reuse Miner plugin. It extracts reuse actions from the applications repositories listed. It generates events for each framework extension found in these applications, analyzing commits in a chronological order and respecting the model in Figure 18. Figure 20 demonstrates the

[3] GitHub Developer API. Available in: <https://developer.github.com/v3/>. Accessed on 31 May 2015.

result of the mining task for the GEF Framework. It shows Shapes application and some of its reuse actions with their dependencies. Each application has a list of commits, ordered in chronological order from top to bottom. The structure in Figure 20 provides all information needed to build the dependency tree (commits sorted in chronological order and dependency information).

4. Generate XES Log: This action is responsible for the generation of the XES logs for the process discovery, using the dependencies trees as input. Different event logs are created for features with more than one trace. In Table 8, even considering only one application, class *EditPartFactory* already has three traces. The event log for this class groups all traces found for all applications (software repositories) mined. Each trace turns into a process trace in the XES event log. The OpenXES library performs the serialization of event logs.

The Eclipse JDT (Java Development Tools) library is used in the identification of classes, methods and interfaces for both the framework and its applications through the creation of ASTs (Abstract syntax tree) for all files in the projects directory. The AST is a tree representation of the source code, where each node corresponds to a construct in code. It has variables, conditional, loops as well as nodes for the declaration of classes, interfaces and methods. Once the AST is created, a visitor class can be used to identify the different types of nodes. Code 9 shows the steps to extract reuse actions from the target framework. For all files in the framework project, an AST is created and the visitor class visits the tree searching for nodes of interest. The result of this parsing is a list of potential reuse actions used in the parsing of framework applications to identify extension points.

Figure 20. Reuse Actions mining for Framework XYZ

Code 9. Algorithm to parse the framework

```
1.  String path = "Framework directory";
2.
3.  // Walk through all paths and files
4.  public void walk(String path) {
5.      File root = new File(path);
6.      File[] files = root.listFiles();
7.      if (list == null) return;
8.      for (File f: list) {
9.          if (f.isDirectory()) {
10.             walk(f);
11.         } else {
12.             String classContent = readFile(f);
13.             getClassInfo(classContent)
14.         }
15.     }
16. }
17.
18. // Generate AST
19. public void getClassInfo(String fileContent) {
20.     ASTParser parser = ASTParser.newParser();
21.     parser.setSource(fileContent);
22.     parser.setKind(AStParser.K_COMPILATION_UNIT);
23.     CompilationUnit ast = parser.createAST();
24.     ast.accept(new FrameworkVisitor());
25. }
26.
27. // Visitor for the generated AST
28. public class FrameworkVisitor extends  ASTVisitor {
29.     // Visit java packages
30.     public boolean visit(PackageDeclaration node) {
31.         // Package node is used to compose the complete class name
32.     }
33.
34.     // Visit classes and interfaces
35.     public boolean visit(TypeDeclaration node) {
36.         // In this method, a new activity to the process is created,
37.         // i.e., a new reuse class for the framework
38.     }
39.     // Visit methods
40.     public boolean visit(MethodDeclaration node) {
41.         // In this method, a new activity to the process is created,
42.         // i.e., a new reuse class for the framework
43.     }
44. }
```

Parsing events from framework applications is not so straightforward, since we need to take ordering into account. Here, instead of having a list of reuse actions, the result of a framework application parsing is a tree structure, the dependency tree. The

identification of reuse actions is performed using the Visitor patterns in Code 9. Next, the information extracted is used as input to populate the dependency tree, as shown in Code 8.

4.3 Process Mining

Once an event log is created from framework applications, the process mining stage can be executed. The ProM framework is used to this end. ProM is an extensible framework created to foment the development of process mining techniques. The algorithms are implemented in the form of plugins, with well-defined inputs and outputs. Figure 21 shows the ProM interface. The left part of the interface manages inputs while the right side manages the outputs. The center of the screen displays a list with the available plugins.

Figure 21. ProM interface

In this dissertation, the following plugins are used: *Filter Log using Simple Heuristics*; *Mine for a Causal Net using Heuristics Miner*; and *Convert C-Net to BPMN*. Figure 22 shows the ordering in which these plugins are applied, indicating the inputs and outputs of each. This process has to be applied to all process and sub-processes,

i.e., all event logs generated in the previous step. Next subsections explain the use of each plugin in Figure 22.

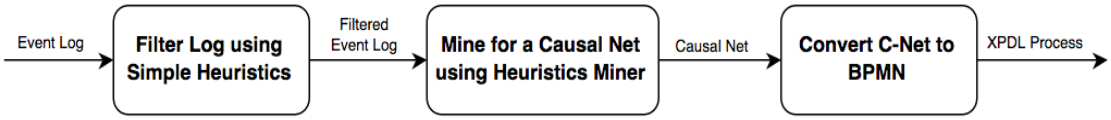


Figure 22. Plugins used for the process mining stage

4.3.1 Filter Log using Simple Heuristics

The first plugin (*Filter Log using Simple Heuristics*) helps removing noise from the event log. Noise behavior may lead to the discovery of spaghetti-like processes that are very unstructured and hard to understand. Finding a structured process is important for the RDL translation, as will be explained in Section 4.4. The log filtering uses the following heuristics:

1. **Event type** → filters events based on the lifecycle status. For this work, all events have the status equals to complete, so this heuristic has no effect in the event log.
2. **Start Event** → shows all activities that appear as start event in the event log and allows the selection of which should remain in the event log. Process instances with no activities selected are removed from the event log.
3. **End Event** → same behavior as explained in heuristic 2 for the end event.
4. **Events frequency** → defines a minimum frequency rate for the events in the event log across process instances. For example, for a frequency rate of 80%, only events that appear in 80% of process instances are included in the filtered event log.

4.3.2 Mine for a Causal Net using Heuristics Miner

This is the main step in the process mining stage since this plugin is the one responsible for discovering a process from the event log. It takes the filtered log, and its

objective is to determine a control-flow that describes the behavior in that log. The algorithm searches for dependencies in the event log to obtain the control flow, also finding conditional paths and loops. Figure 23 show the ProM interface for the use of the heuristics miner plugin. It expects an event log as input (left side of the screen) – in our work, the event logs mined with the Event log miner. After providing the input, ProM displays the screen in Figure 24 to configure the algorithm parameters. We use all default parameters and select the option to have unique process start and end. Figure 25 shows an example of a causal net generated by this plugin, using information from the Shapes and Logic applications.

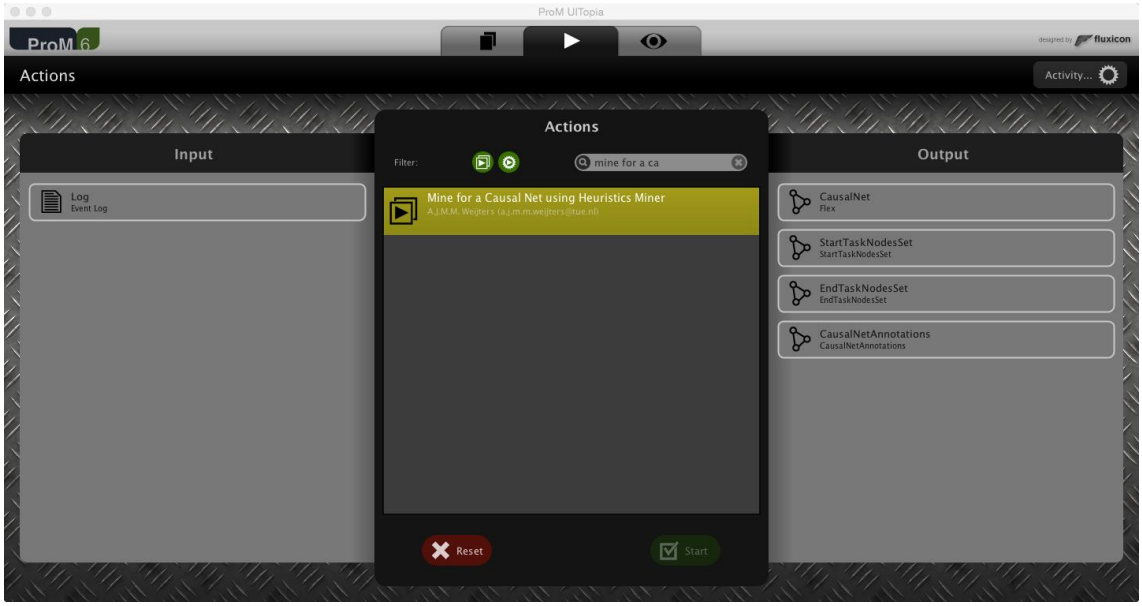


Figure 23. ProM Framework – Mine for a Causal Net using Heuristics Miner plugin.

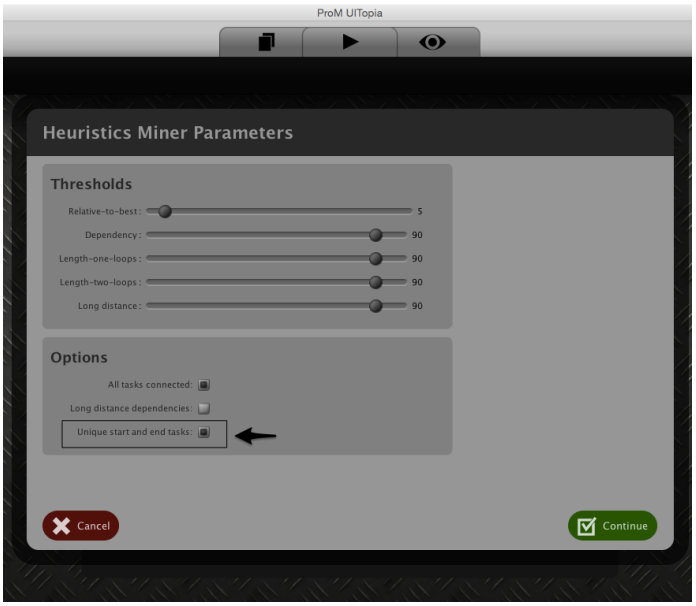


Figure 24. Heuristics Miner parameters

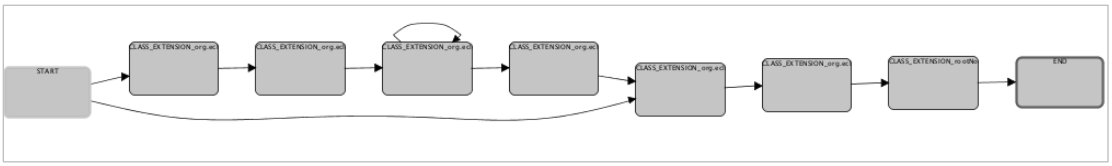


Figure 25. Causal-net mined for GEF based on Shapes and Logic applications

4.3.3 Convert C-Net to BPMN

Finally, the *Convert C-Net to BPMN* plugin is used to facilitate the next phase, the translation of the discovered process to RDL. The ProM framework offers several plugins to convert from one process representation to another. In our case, the causal net from the previous plugin is converted to BPMN and exported from ProM in XPDL.

XPDL (XML Process Definition Language) provides a way of serializing BPMN process, including all aspects of BPMN process definition notation. Using XPDL it is possible to describe all process activities, the process control-flow and also include graphical information on how the process should be displayed (WFMC, 2012).

Figure 26 shows the result of the conversion of the causal-net in Figure 25 to BPMN. Observing the processes depicted in Figure 25 and Figure 26, we can observe

the occurrence of a loop construct and a conditional. In the BPMN process, these constructs are delimited by gateways. The loop is represented by gateways G2 and G4 – a choice has to be made on gateway G4 to either repeat the previous activity (6) or continue to the next one (8). The conditional is found on gateway G1 and, depending on the user's choice, part of the process execution may be skipped (path from activity 3 to activity 8).

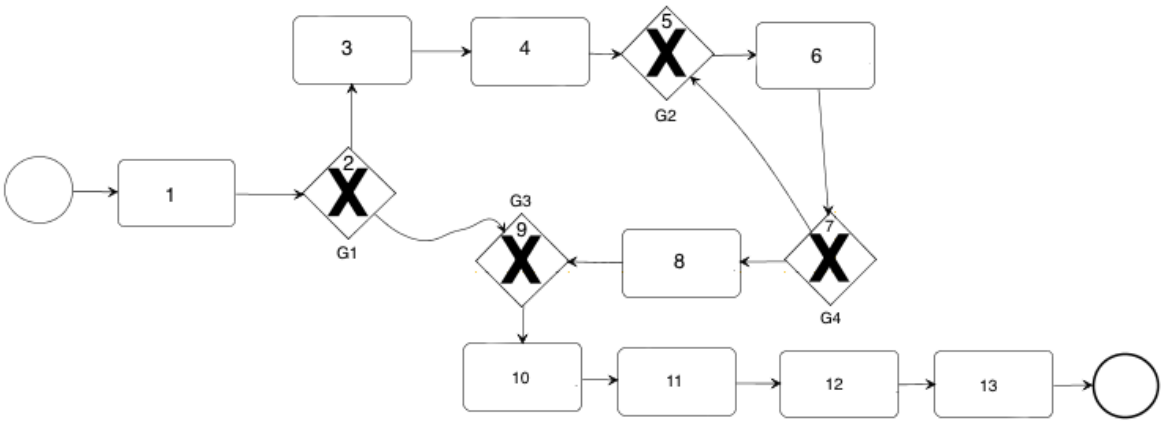


Figure 26. BPMN Process for Framework XYZ

Section 4.4 explains in details how the XPDL representation of this process is translated to RDL.

4.4 RDL Generation

RDL is a procedural language that can be used to describe the reuse process of frameworks. To this end, RDL has two types of commands: commands for reuse activities and commands to express the control-flow of the process. The first type comprises the manipulation of framework elements (e.g.: `CLASS_EXTENSION`, `METHOD_EXTENSION`) while the second one indicates the sequence of activities, the possibility of repetition (`LOOP`) and choices (`IF`).

In order to represent a XPDL process using RDL, the aforementioned RDL commands need to be identified in the process. The commands to manipulate the

framework are the activities in the process and have a straightforward translation. Sequences do not have special commands and also have an immediate transformation. Loops and ifs, however, need a particular treatment. These structures will be handled with the assistance of RPSTs (Refined Program Structure Trees).

JOHNSON *et al.* (1994) developed an algorithm to decompose a control-flow graph into single-entry single-exit (SESE) regions. These regions can be organized hierarchically in a tree called the Program Structure Tree (PST), where nodes represent SESE regions and edges the nesting of regions. PSTs are used in the literature for different purposes. Originally, it was developed as a tool to improve the performance of program analysis algorithms. Later, this technique was also used in the context of business processes, considering that the control flow of these processes can be modeled as a workflow graph. Other uses of PSTs include the conversion between process representations (e.g.: BPMN to BPEL) (VANHATALO *et al.*, 2008), the verification of process soundness (VANHATALO *et al.*, 2007) and the syntactic categorization of program blocks. In this work, we are interested in the latter use, the categorization of program blocks, finding sequences, ifs, repeat-until, etc. The RPST proposed in (VANHATALO *et al.*, 2008) is used for this purpose.

The RPST allows a unique, and modular decomposition of the workflow graph. Moreover, the RPST defines the regions limits (entry and exit) as nodes in opposition to edges, as originally proposed. This change leads to finer blocks, allowing the analysis of smaller blocks, which can have some advantages (e.g.: debugging smaller blocks is easier than searching for errors in larger ones).

Figure 27 exemplifies the identification of sub-workflows in a process, according to its syntactical category. In this process, it is possible to identify sequences (nodes 10 to 13, for example), a loop (nodes 5, 6 and 7) and a conditional on node 2 from which there is a choice between two different paths. Figure 28 shows the RPST for this

process. For each tree level, an indication of the syntactical category that connects the nodes in that level is given. For instance, the two nodes in the last level are in a sequence. Each node is a region, a polygon (P_n) or a bond (B_n), according to its structure. All regions indicate its entry node (number on the left) and its exit node (number on the right).

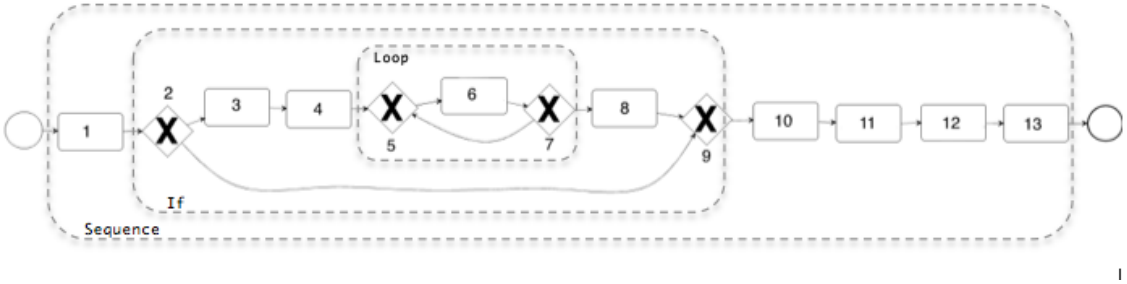


Figure 27. Process blocks example

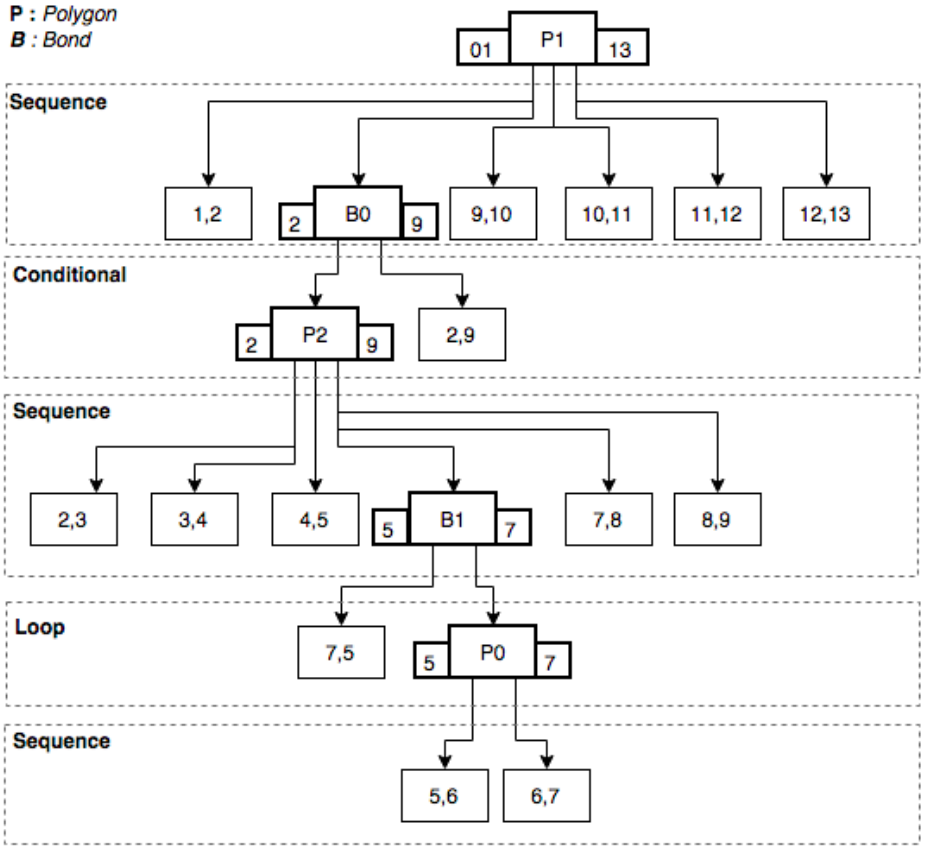


Figure 28. RPST for the process in Figure 27.

Once the RPST is ready, it is traversed to generate the RDL script. First, all nodes in a region are sorted according to entry node and exit node. The RPST in Figure 28 is already sorted. For example, we can observe that nodes on level 2 – level 1 is the tree root – are ordered by entry and exit nodes (1,2 → 2,9 → 9,10 → 10,11 → 11,12 → 12,13). The following heuristic are defined for generating the RDL script:

1. For a given level, if all nodes have the same entry and exit nodes, this corresponds to a conditional;
2. For a given level, if all nodes have entry and exit nodes in a sequence, this corresponds to a sequence;
3. For a given level, if the entry node of the first node is equal to the exit node of the last node in that level, this corresponds to a loop.

It is worth noting that the process must be well formed otherwise the algorithm for generating the RPST will not have a deterministic result. A well-formed BPMN must respect the semantics defined by each element. For gateways, for example, it is important to have well-defined regions for merges and splits, which means that these regions have an explicit gateway in its beginning and an explicit gateway to finish it. This is not always the case when using process discovery algorithms, mainly for spaghetti-like processes or processes that simply represent all possible paths in the event log.

The implementation of the algorithm to find RPST used in this work is the one present in the jBPT code present in the jBPT code library, a library that assembles a compendium of process analysis techniques analysis techniques (jbpt, 2013). The algorithm for traversing the RPST is shown in Code 10. Every trivial node (node that does not have children) in the RPST corresponds to an edge.

Code 11 shows the script version generated when executing this algorithm over the RPST in Figure 28. Column *Final Script* in this table indicated the final script generated by traversing the RPST tree while column *Edges* indicate the edges involved in each step.

Code 10. RPST traversal algorithm

```
1.    traverseFromNode(root) {
2.        // Node does not have a children
3.        if (root.isTrivial()) {
4.            printNode(root);
5.            return;
6.        }
7.
8.        // Children nodes follow heuristic 3
9.        if (root.isLoop()) {
10.           printLoopCommand()
11.       }
12.
13.       // Children nodes follow heuristic 2
14.       if (root.isFirstConditional()) {
15.           printIfCommand()
16.       }
17.
18.       // Children nodes follow heuristic 2 and
19.       // it is not the first condition
20.       if (root.isConditional()) {
21.           printElseIfCommand()
22.       }
23.
24.       children = root.getChildren()
25.       for (childNode in children) {
26.           traverseFromNode(childNode)
27.       }
28.
29.       printCloseBlock();
30.   }
```

Code 11. Script for process in Figure 27

Final Script	Edges

1. {
2. 1;
3. if() {
4. 3;
5. 4;
6. loop() {
7. 6;
8. }
9. 8
10. }
11. 10;
12. 11;
13. 12;
14. 13;
15. }
	1. {
	2. [1,2];
	3. [2,9]; [Conditional]
	4. [2,3] – [3,4];
	5. [3,4] – [4,5];
	6. [5,7]; [Loop]
	7. [5,7] – [6,7]
	8. [end loop]
	9. [7,8] – [8,9];
	10. [end conditional]
	11. [9,10]-[10-11];
	12. [10,11] – [11,12];
	13. [11,12] – [12,13];
	14. [12,13];
	15. }

5 Validation

In this chapter, the solution presented in Chapter 4 will be applied using two frameworks: GEF framework, already used as example in previous chapters, and Graphiti framework, a second graphical framework from Eclipse platform. The processes discovered for each of these frameworks are presented, as long as an evaluation of them in terms of precision and recall.

5.1 Process Mining Evaluation

In Chapter 4, a solution to discover the instantiation process from application code was presented. This solution includes the extraction of events from code, the generation of event logs, and the application of process discovery algorithms to these events logs with the subsequent translation of XPDL processes to RDL. In this Chapter, the whole solution is applied to two frameworks: 1) GEF framework, the Eclipse graphical framework; 2) Graphiti, a graphical framework based on GEF. In next subsections we present the mining results for a subset of frameworks applications for each framework. In addition, we present the evaluation of the results in terms of precision and recall.

Precision and recall are metrics commonly used to evaluate data mining results. Consider a query, its expected results and the real results for the query. Precision evaluates how many relevant results were retrieved from the whole set of results. Recall, in turn, measures how many relevant instances were retrieved (AALST, 2011). These metrics are summarized below in Equation 2 and Equation 3.

$$Precision = \frac{t_p}{p}$$

$$t_p \rightarrow number\ of\ relevant\ results\ returned$$

$$p \rightarrow number\ of\ results\ returned$$

Equation 2. Precision metric

$$Recall = \frac{t_p}{p'}$$

$$t_p \rightarrow number\ of\ relevant\ results\ returned$$

$$p' \rightarrow number\ of\ relevant\ results\ expected$$

Equation 3. Recall Metric

In order to calculate both metrics, we need a reference to provide the basis for calculating the number of relevant results. For this reason, in this Chapter, RDL scripts were manually created based on the frameworks online documentation. These scripts provide the list of relevant activities that should be returned by the mined process. Moreover, we want to take conditionals and loops into consideration to compare the mined process and the one that was manually created. To include this information in the calculation of precision and recall, we include a penalty to the equations and we can rewrite them as in Equation 4 and Equation 5. If one activity is found in the mined process, but it has a different cardinality than in the original process, it is included in the calculation multiplied by the penalty factor of 0.5. By different cardinality we mean any non-matching combination between mandatory activities, conditional activities and repeatable activities.

$$Precision = \frac{t_p + 0.5 \times h_p}{p}$$

$t_p \rightarrow relevant\ results\ returned\ that\ have\ an\ exact\ match\ to\ the\ reference\ model$

$h_p \rightarrow relevant\ results\ that\ do\ not\ match\ the\ reference\ model\ perfectly$

$p \rightarrow number\ of\ results\ returned$

Equation 4. Precision metric considering possible differences in the results.

$$Recall = \frac{t_p + 0.5 \times h_p}{p'}$$

$t_p \rightarrow relevant\ results\ returned\ that\ have\ an\ exact\ match\ to\ the\ reference\ model$

$h_p \rightarrow relevant\ results\ that\ do\ not\ match\ the\ reference\ model\ perfectly$

$p' \rightarrow number\ of\ relevant\ results\ expected$

Equation 5. Recall metric considering possible differences in the results.

In next sections, we present the processes discovered by the Reuse Miner for Graphiti – Section 5.2– and GEF Framework – Section 5.3. Both frameworks are Eclipse-based frameworks and their applications are implemented as Eclipse plugins. For this reason, there are some actions that the user must perform during the application development that cannot be automated by the ReuseTool. These tasks are included in the manually created RDL using the command *external_task*, but they are not included in the calculation of precision and recall since they are not included in the mined event logs.

5.2 Graphiti

We start our evaluation using Graphiti framework because it is a simpler framework when compared to GEF. Graphiti framework is another Eclipse framework to create

graphical editors. It uses GEF and Draw2D frameworks and supports the use of EMF for modeling the editor domain. Graphiti exposes a JAVA API for developers and no knowledge of GEF and Draw2D is necessary.

Based on the information provided by the online help[4], we manually created a RDL script, split into 3 recipes: *Abstract Diagram Type Provider*, *Default Feature Provider* and *Main.* The recipe for *Default Feature Provider* is not included in this chapter because it is a long script (77 lines of code), but it can be found in Appendix A. Table 9 depicts the other two processes, also included in Appendix A. The Main recipe delineates the main path of the process, calling other recipes if needed. The recipe *AbstractDiagramTypeProvider* represents the steps to instantiate the framework class with the same name. This class is one of the main Graphiti classes.

In this Section, we will evaluate the Reuse Miner results using two scenarios. First, in Section 5.2.1, we use only one application, the Tutorial Application distributed with the framework. This application is the result of the tutorial used as basis for creating the reference process. Next, Section 5.2.2 presents the result of executing the Reuse Miner approach using four Graphiti applications as input.

Table 9. RDL script manually created for Graphiti framework

```
1.  cookbook Graphiti
2.  recipe Main {
3.     external_task ("Create an eclipse plugin");
4.     pack = new_package(FrameworkModel, "?");
5.     // call recipe
6.     AbstractDiagramTypeProvider();
```

[4] Graphiti Online Help. Available in <http://help.eclipse.org/kepler/index.jsp?nav=%2F27>. Accessed on May 29, 2015

```
7.          external_task("Register the diagram in plugin.xml");
8.
9.          loop ("Create image provider?") {
10.            class_extension("org.eclipse.graphiti.ui.platform.
11.                        AbstractImageProvider", "?");
12.            external_task("Register extension point in
13.            plugin.xml");
14.         }
15.
16.         loop ("Contribute to eclipse's property feature?") {
17.           class_extension("org.eclipse.graphiti.ui.platform.
18.                       AbstractPropertySectionFilter", "?");
19.
20.           loop ("Create property section?") {
21.              class_extension("org.eclipse.graphiti.ui.platform.
22.                        GFPropertySection", "?");
23.           }
24.           external_task("Register extension point in plugin.xml");
25.        }
26.     }
27.     recipe AbstractDiagramTypeProvider {
28.        // Creates a default Feature Provider
29.        DefaultFeatureProvider();
30.
31.        // Creates a default tool behavior Provider
32.        class_extension("org.eclipse.graphiti.tb.
33.                       DefaultToolBehaviorProvider", "?");
34.
35.        // Creates a diagram type
36.        class_extension("org.eclipse.graphiti.dt.
37.                       AbstractDiagramTypeProvider", "?");
38.     }
        …
```

5.2.1 Tutorial Application

This Section presents the processes mined for the Tutorial application, the application generated by following the steps of the same tutorial used to create the RDL in Table 9. The comparison between the manually created script and the mined one is performed in terms of precision and recall, as previously explained in Section 5.1. Table 11 depicts in details the results of this comparison for the main recipe. Columns *Reference* and *Mined* in Table 10 indicate the process structure associated with the process activity. When an activity exists in both scripts, but a different process structure was

found after mining, we use the weight of 0.5 to take this difference into account for the values of precision and recall, as shown in Equation 4 and Equation 5.

Table 10. Precision and Recall for Main recipe

ACTIVITY NAME	*REFERENCE*	*MINED*	*WEIGHT*	*CORRECTED WEIGHT*
AbstractDiagramTypeProvider	X	X	1	1
AbstractImageProvider	Loop	Loop	1	1
AbstractPropertySectionFilter	Loop	Loop	1	1
GFPropertySection	Loop	If	0.5	0.5
PredefinedColoredAreas	-	X	0	1
	Precision		0.7	0.9
	Recall		0.875	0.9

Mined activities not found in the original RDL script are analyzed in two steps. First, they are considered to be wrong and receive a weight of 0 for the calculus of precision and recall. This results in the column *Weight* in Table 14. Next, we evaluate the new activities to decide if they are actually incorrect or if they could be added to the reference RDL. In Table 10, the activity *PredefinedColoredAreas* was implemented for styling purposes, which represents an optional feature of the framework. For this reason, the original script could be updated with the inclusion of this activity and we included this information for computing the value of precision and recall in the last column of Table 10 (*Corrected Weight*). The difference found for class *GFPropertySection* (conditional instead of a loop) is explained by the low generalization of the mined process due to the use of a single application as input.

If we didn't take the process differences into account for precision and recall metrics and considered only the set of activities, we would have the results in Table 11, which considers Equation 2 and Equation 3. This result does not represent what is

happening in reality, since the mined process contains discrepancies when compared to the reference process.

Table 11. Precision and Recall for mined processes

	PRECISION	***RECALL***
Main	$\frac{relevant}{total} = \frac{3}{3} = 1$	$\frac{relevant}{returned\ relevant} = \frac{3}{3} = 1$
Abstract Diagram Type Provider	$\frac{relevant}{total} = \frac{4}{5} = 0.8$	$\frac{relevant}{returned\ relevant} = \frac{4}{4} = 1$

Next, we compute the values of precision and recall for the remaining recipes found for the Tutorial application: *DefaultFeatureProvider class*, responsible for delivering the features or operations that can be performed over model elements; and *AbstractDiagramTypeProvider* class, the class that manages the interactions between features and the diagram. The results of precision and recall are presented in Table 12. The process mined for *AbstractDiagramTypeProvider* is identical to the reference process, which explains the high values for precision and recall. For the DefaultFeatureProvider, we can make a few considerations about the lower values:

- Some features have a default implementation, such as the Remove and Delete features, and the developer should decide if they need a custom implementation. The tutorial application do not include the mentioned features and that explains the difference between p and p'. The last line in Table 12 shows the recalculated value for recall considering the removal of these activities from the reference process. Recall improves a little as a result of the reduction in the number of expected activities;
- Since we are dealing with a single application, it is harder to mine conditional features since we are limited to what happened in this unique application. In addition, we can't mine most of the loops that were expected because most

features are implemented only once. This resulted in the high value in column h_p (number of activities mined with a different process structure).

Table 12. Precision and Recall for DefaultFeatureProvider and AbstractDiagramTypeProvider

	p	p'	t_p	h_p	PRECISION	RECALL
AbstractDiagramTypeProvider	3	3	3	0	1.0	1.0
DefaultFeatureProvider	15	17	2	13	0.567	0.5
DefaultFeatureProvider (2)	15	15	2	13	0.567	0.567

In Section 5.2.2, we have the results obtained for the same processes when we have more input applications.

5.2.2 Multiple applications

This Section presents the results of mining an RDL process for Graphiti framework using four framework applications. These applications include two example applications (Tutorial and Chess) that come with Graphiti and two applications found using GitHub search API. The API returned a longer list of results, but most of them consisted of duplicated applications based on the Tutorial example and incomplete applications. Table 13 shows the list of applications (first line) and the list of sub-processes (first column). The X in Table 13 indicates the existence of a feature (sub-process) in an application.

Table 13. Sub-processes mine for Graphiti based on the input applications

	TUTORIAL	PERMET	SKETCHER	CHESS
org.eclipse.graphiti.dt.AbstractDiagramTypeProvider	X	X	X	X
org.eclipse.graphiti.ui.features.DefaultFeatureProvider	X	X	X	X
org.eclipse.graphiti.tb.DefaultToolBehaviorProvider	X	X	0	X

Table 14. Precision and Recall for main recipe

ACTIVITY NAME	*REFERENCE*	*MINED*	*WEIGHT*	*CORRECTED WEIGHT*
AbstractDiagramTypeProvider	Loop	Loop	1	1
AbstractImageProvider	Loop	Loop	1	1
AbstractPropertySectionFilter	Loop	Loop	1	1
GFPropertySection	Loop	If	0.5	0.5
PredefinedColoredAreas	-	If	0	1
AbstractCreateConnectionFeature	-	If	0	0
	Precision		0.583	0.75
	Recall		0.875	0.9

Table 14 shows the comparison between the RDL script – main recipe – used as reference and the mined one. Mined activities not found in the original RDL script are analyzed in two steps. First, they are considered to be wrong and receive a weight of 0 for the calculus of precision and recall. This results in the column *Weight* in Table 14. Next, we evaluate the new activities to decide if they are actually incorrect or if they could be added to the original RDL. In Table 14, we have the following cases:

- **Activity *PredefinedColoredAreas***: as aforementioned in Section 5.2.1, this activity was implemented for styling purposes, which represents an optional feature of the framework. For this reason, the original script could be updated with the inclusion of this activity.
- **Activity *AbstractCreateConnectionFeature*:** according to the framework documentation, this class should be delivered by the *DefaultFeatureProvider* so it should not be found in the main process, but in the *DefaultFeatureProvider* process. This may indicate a problem in one of the input applications. It is worth mentioning that the entire process structure from lines 5-12 in Process 4, more specifically the if-then-else structure, only exists because of this activity. Filtering

the occurrence of class *AbstractCreateConnectionFeature* would result in better mining outcomes.

The last column in Table 14 presents the values for precision and recall considering the inclusion of the activity *PredefinedColoredAreas*.

Table 15 contains the values of precision and recall for the remaining sub-processes mined from Reuse Miner. The *DefaultToolBehavior* sub-process was not included in the table because no sub-process was created for it manually, so there is no reference process to compare. Again, the last line of Table 15 displays the values of precision and recall for *DefaultFeatureProvider* considering also the removal of activities *DefaultRemoveFeature* and *DefaultDeleteFeature* that are optional and already have a default implementation. With this, we achieve a higher recall value.

Table 15. Precision and recall for AbstractDiagramTypeProvider and DefaultFeatureProvider

	p	p'	t_p	h_p	PRECISION	RECALL
AbstractDiagramTypeProvider	3	3	2	1	0.833	0.833
DefaultFeatureProvider	15	17	6	10	0.733	0.647
DefaultFeatureProvider	15	15	6	10	0.733	0.733

In Table 16, we have a comparison of the results obtained. A few considerations about these results:

- The Main recipe using 4 applications contained an unexpected activity, which resulted in the lower precision.
- The *AbstractDiagramTypeProvider* mined using four applications pointed the activity *DefaultToolBehaviorProvider* to be optional, but the reference model list this activity as mandatory. According to the documentation, this class is used to add functionality to existing editing concepts of Eclipse workbench. In this case, this activity may be considered optional since it is not essential for the editor

implementation. If we change the reference script, we have precision and recall for the 4 apps execution equals to 1, and the values for the single application execution changes to 0.833.

- In the case of the *DefaultFeatureProvider* class, we have an improved result of precision and recall when using more applications. This process is composed of a list of framework classes that can be extended in the creation of a custom editor. Many of these features may be repeated or skipped (optional activities). The improved values of precision and recall here shows an increased generalization of the mined process, considering that more conditionals and repetitions could be found when we used more applications.

Table 16. Comparison of the mined values using one and four applications

	1 APP		4 APPS	
	PRECISION	RECALL	PRECISION	RECALL
Main	0.9	0.9	0.75	0.9
AbstractDiagramTypeProvider	1.0	1.0	0.833	0.833
DefaultFeatureProvider	0. 567	0. 567	0.733	0.733

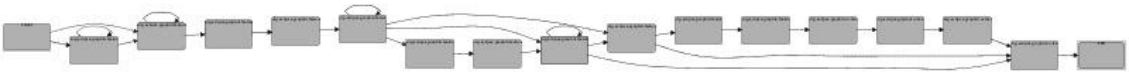
5.2.3 Mined processes

The mined causal-net and their corresponding RDL scripts using the applications in Table 13 are depicted below from Process 1 to Process 4, including the sub-processes listed in Table 13 and the main process. It is worth mentioning that the final process needs to be complemented with the messages that should be presented to the developers when ReuseTool executes a conditional or a loop (e.g., the conditional in Process 1 line 2).

Process 1. Process for AbstractDiagramTypeProvider


```
1.  recipe AbstractDiagramTypeProvider {
2.    if ()? {
3.      class_extension(
4.          "org.eclipse.graphiti.tb.DefaultToolBehaviorProvider", "?");
5.    }
6.    DefaultFeatureProvider();
7.    class_extension("org.eclipse.graphiti.dt.Abstract
8.                    DiagramTypeProvider", "?");
9.  }
10.
```

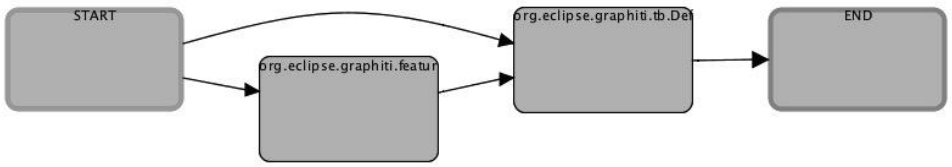
Process 2. Process for DefaultFeatureProvider



```
1.  recipe {
2.    if ()? {
3.      loop() {
4.        class_extension("org.eclipse.graphiti.features.
5.                        custom.AbstractCustomFeature", "?");
6.      }
7.    }
8.    loop() {
9.      class_extension("org.eclipse.graphiti.features.
10.                     impl.AbstractAddFeature", "?");
11.   }
12.   class_extension("org.eclipse.graphiti.features.impl.
13.                   AbstractAddShapeFeature", "?");
14.   class_extension("org.eclipse.graphiti.features.impl.
15.                   AbstractCreateConnectionFeature", "?");
16.   loop() {
17.     class_extension("org.eclipse.graphiti.features.impl.
18.                     AbstractCreateFeature", "?");
19.   }
20.   loop() {
21.     class_extension("org.eclipse.graphiti.features.impl.
22.                     AbstractUpdateFeature", "?");
23.   }
24.   if ()? {
25.     class_extension("org.eclipse.graphiti.features.impl.
26.                     AbstractDirectEditingFeature", "?");
27.     class_extension("org.eclipse.graphiti.features.impl.
28.                     AbstractLayoutFeature", "?");
29.   }
30.
31.   class_extension("org.eclipse.graphiti.features.impl.
32.                   DefaultMoveShapeFeature", "?");
33.
34.   if ()? {
35.     class_extension("org.eclipse.graphiti.features.impl.
36.                     DefaultReconnectionFeature", "?");
37.     class_extension("org.eclipse.graphiti.features.impl.
```

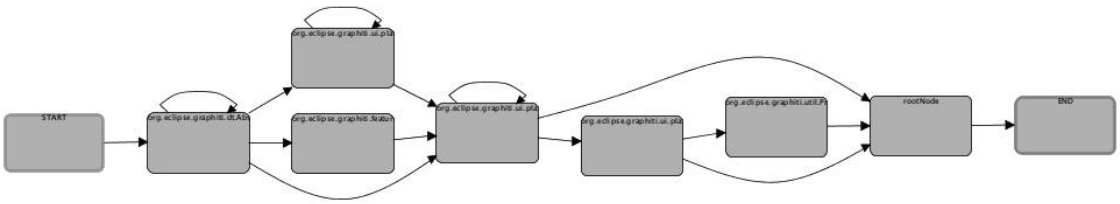
```
38.                     DefaultResizeShapeFeature", "?");
39.
40.         class_extension("org.eclipse.graphiti.ui.features.
41.                     AbstractCopyFeature", "?");
42.         class_extension("org.eclipse.graphiti.ui.features.
43.                     AbstractDrillDownFeature", "?");
44.         class_extension("org.eclipse.graphiti.ui.features.
45.                     AbstractPasteFeature", "?");
46.      }
47.
48.      class_extension("org.eclipse.graphiti.ui.features.
49.           DefaultFeatureProvider", "?");
50.   }
```

Process 3. DefaultToolBehavior process




```
1.    recipe {
2.       if ()? {
3.          class_extension("org.eclipse.graphiti.features.custom.
4.                     AbstractCustomFeature",  "?");
5.       }
6.       class_extension("org.eclipse.graphiti.tb.DefaultToolBehaviorProvider",  "?");
      }
```

Process 4. Graphiti Main process




```
1.    recipe {
2.       loop() {
3.          class_extension("org.eclipse.graphiti.dt.
4.                     AbstractDiagramTypeProvider", "?");
5.       }
6.       if ()? {
7.          loop() {
8.             class_extension("org.eclipse.graphiti.ui.platform.
9.                     AbstractImageProvider", "?");
10.         }
11.      } else if () {
12.         class_extension("org.eclipse.graphiti.features.impl.
13.                     AbstractCreateConnectionFeature", "?");
14.      }
15.      loop() {
16.         class_extension("org.eclipse.graphiti.ui.platform.
```

```
17.                     AbstractPropertySectionFilter", "?");
18.     }
19.     if ()? {
20.       class_extension("org.eclipse.graphiti.ui.platform.
21.                     GFPropertySection", "?");
22.       if ()? {
23.         class_extension("org.eclipse.graphiti.util.
24.                       PredefinedColoredAreas", "?");
          }
        }
      }
```

5.3 GEF (Graphical Editing Framework)

As aforementioned, GEF is an Eclipse framework for the development of custom graphical editors and views. GEF applications are based on the MVC (model-view-controller) architecture: the developer defines the model and, through the extension of framework classes, she creates controllers that manipulate domain elements and map these elements to their respective views. Controllers in GEF are implemented through the extension of Edit Parts and the framework provides a plethora of base implementations for this purpose. Likewise, the framework offers several edit policies to provide editing capabilities to edit parts. For this reason, the framework contains a wide variety of hotspots. In this section, we only consider one small subset of hotspots. Five applications were used in the mining task: three GEF example applications (Flow, Logic and Shapes) and two applications found using GitHub API (smooks-editor and Polygon).

Table 17. List of GEF classes found in all five applications

FEATURE LIST
org.eclipse.gef.EditPartFactory
org.eclipse.gef.editparts.AbstractConnectionEditPart
org.eclipse.gef.editparts.AbstractGraphicalEditPart

Table 17 lists the features considered in this section. All five applications mentioned implemented these features. The reference RDL will be created from the application repositories, observing the dependencies in code and writing a script that represents them. For EditPartFactory and AbstractConnectionEditPart, we achieved a perfect match for the list of activities that should be found, the conditionals and the loops. On the other hand, the class AbstractGraphicalEditPart did not return a well-structured process, so RPST algorithm does not find a deterministic result. The traces in AbstractGraphicalEditPart's event log are completely different from one another and the process discovery couldn't generalize the behavior. In this case, the discovered process corresponds to all possible paths found in the event log. Table 18 shows the list of activities and in which traces they can be found. The resulting process contains all possible paths described in the event log. If we had similar traces we could filter rare activities and execute the process discovery again, but from Table 18 we can see that all activities have low frequencies.

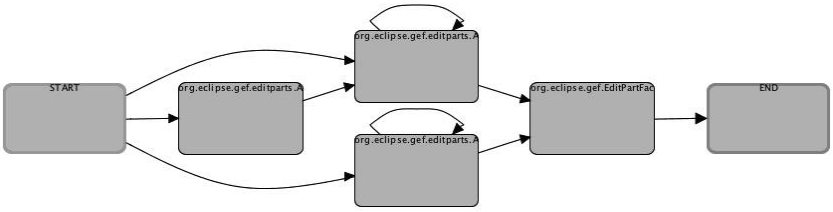
Section 5.3.1 shows the mined processes. For *EditPartFactory* and *AbstractConnectionEditPart,* it shows the mined RDL processes as well in Table 19 and Table 20, respectively. The manual processes generated for *EditPartFactory* and *AbstractConnectionEditPart* can be found in Appendix B. In this case, the manual processes were created based on the code repositories and the discovered processes had a perfect match. A limitation in this evaluation is the lack of a specialist to assess the quality of these processes, both the manually created and the mined one. Example applications distributed with GEF framework were included in the list of repositories as examples of correctly created applications in order to mitigate this limitation.

Table 18. Traces in AbstractGraphicalEditPart

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
org.eclipse.gef.tools.CellEditorLocator										X	X									
org.eclipse.gef.editpolicies.ComponentEditPolicy						X	X	X	X	X										
org.eclipse.gef.commands.Command								X												
org.eclipse.gef.editpolicies.ContainerEditPolicy													X							
org.eclipse.gef.tools.DirectEditManager										X	X									
org.eclipse.gef.editpolicies.DirectEditPolicy										X										
org.eclipse.gef.editpolicies.FlowLayoutEditPolicy														X						
org.eclipse.gef.editpolicies.GraphicalEditPolicy			X											X						
org.eclipse.gef.editpolicies.GraphicalNodeEditPolicy						X	X													
org.eclipse.gef.editpolicies.LayoutEditPolicy													X							
org.eclipse.gef.editpolicies.SelectionEditPolicy												X								
org.eclipse.gef.editpolicies.XYLayoutEditPolicy	X	X	X	X	X															
org.eclipse.gef.editparts.AbstractGraphicalEditPart	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

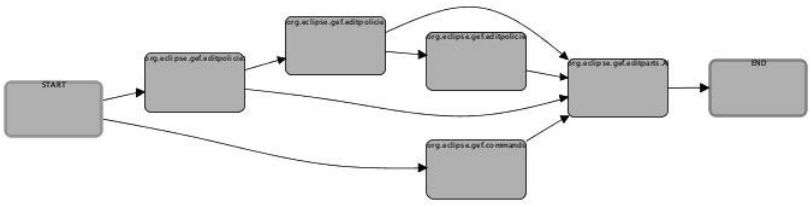
5.3.1 Mined Processes

Table 19. Mined process for EditPartFactory interface




```
1.    recipe {
2.
3.      if ()? {
4.        loop() {
5.          class_extension("org.eclipse.gef.editparts.
6.
7.                        AbstractTreeEditPart", "?");
8.        }
9.      } else if () {
10.       if ()? {
11.         class_extension("org.eclipse.gef.editparts.
12.
13.                       AbstractConnectionEditPart", "?");
14.
15.       }
16.       loop() {
17.
18.         class_extension("org.eclipse.gef.editparts.
19.                       AbstractGraphicalEditPart", "?");
20.       }
21.
22.   }
      new_class(pack, , "?");
      new_interface_realization("org.eclipse.gef.EditPartFactory", "?");
    }
```

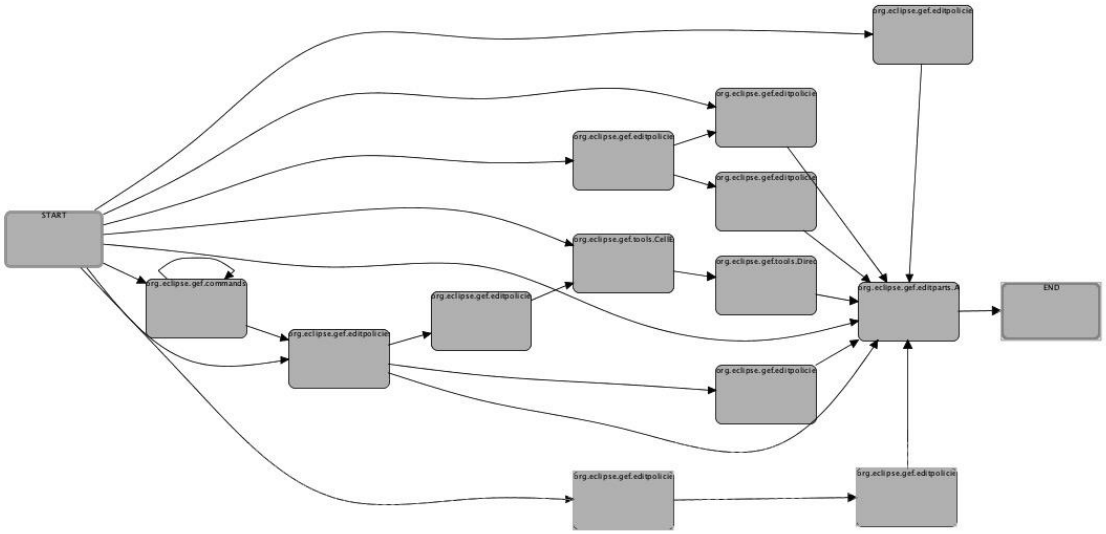
Table 20. Process mined for AbstractConnectionEditPart class




```
1.    recipe {
2.      if ()? {
3.
4.        class_extension("org.eclipse.gef.editpolicies.ConnectionEditPolicy", "?");
5.        if ()? {
6.          class_extension("org.eclipse.gef.editpolicies.
7.                        ConnectionEndpointEditPolicy", "?");
8.
9.          if ()? {
```

```
10.              class_extension("org.eclipse.gef.editpolicies.
11.                      BendpointEditPolicy", "?");
12.            }
13.          }
14.      } else if () {
15.          class_extension("org.eclipse.gef.commands.Command", "?");
16.      }
17.      class_extension("org.eclipse.gef.editparts.
18.              AbstractConnectionEditPart", "?");
19.  }
```

Process 5. Process mined for AbstractGraphicalEditPart.



5.4 Threats to Validity

This Section discusses threats to validity related to the evaluation presented in this chapter. The manual creation of the RDL scripts represents a threat to the validity of the evaluation results presented in this chapter. The scripts were created based on the official framework documentation and application repositories in order to minimize the risk of having a bad reference script. The fact that most activities in the reference script matched the activities in the mined script, using different applications than the one used to generate the reference process, may indicate that the reference process contains a suitable subset of framework activities. The selection of framework

application is also a threat and to minimize it example applications distributed with the frameworks were included in the evaluation set.

When dealing with process mining, a higher number of traces in the event log may result in a better generalization of the process at hand. In this case, our processes may overfit the behavior from our sample applications. This can be observed in Process 5. The input event log for mining this process had traces very different from one another and this resulted in a process that is not generic, but a process that contains all possible paths found in the event log. The use of more framework applications could improve these results, by offering more traces that could help generalizing the process behavior.

In terms of the calculation of precision and recall, the use of the penalty factor is a threat to validity since it impacts directly in the final results of precision and recall. A simpler way to calculate the values of precision and recall uses Equation 2 and Equation 3. However, using these equations, the correct activities returned would not be considered. The role of the penalty factory is to punish the difference while still considering the correct activity.

It is worth mentioning that we use metrics of precision and recall because we did not take the ordering between commands in the final discovered process into consideration. Although ordering information is of utmost importance for discovering the process behavior, once the recipes are mined, we have no indications of what activities could be parallel and which should really occur in a sequence. RDL has commands to indicate parallel activities and this will be handled in a future work.

6. Conclusion

On this chapter, we present the final considerations about this dissertation. In addition, we discuss about some limitations of the solution presented and future work.

6.1. Conclusions

Software reuse has a central role in the software development process with the main objective of increasing software quality along with reduced development time. One of the main existing technologies to achieve these goals are object-oriented frameworks that aim at improving software reuse for applications of a certain domain by providing the core implementation for these applications and offering flexible parts that should be customized to meet developers' requirements. Although this flexibility allows the construction of a variety of applications using the same framework, it also impacts on its understandability. For this reason, having good learning resources is crucial for the success of the framework.

The great amount of information about the development of framework applications stored in software repositories fostered the use of data mining techniques over these repositories to gather information that could be used to support the development of new applications. In this dissertation, we presented an approach that uses this information to generate event logs and execute process mining. The final outcome is an RDL process that not only describes the instantiation process, but it can also provide ReuseTool with the steps it needs to semi-automate the instantiation process.

In Chapter 5, Reuse Miner discovered processes for two frameworks: Graphiti and GEF. Graphiti framework is a framework built on top of the GEF framework and intends to provide simpler flexible points for the development of graphical editors in comparison to GEF. We could mine processes for Graphiti's main hotspots, obtaining good results for precision and recall in general. The number of applications used as input impacts on these metrics when the discovered process is expected to have loops and conditionals due to the fact that the more applications we have the better we will be able to generalize the process behavior. In addition, it is important to select good repositories as input, otherwise the final process will have noisy behavior, unless this noise is correctly filtered out of the event logs before the process discovery phase.

GEF Framework offers a wider variety of hotspots and we focused in three of them for the analysis in Chapter 5. Two of these processes have a perfect match to the process build manually, which was expected since the manual process used the repositories as reference. Nonetheless, the process discovery algorithm could not generalize the behavior for the third process based on the information in the event log. All traces in this event log are different from one another and the outcome in this situation is a process that depicts all possible paths found in the event log. The result of the mining task is a spaghetti-like process that cannot be translated to RDL using Reuse Miner's approach, as shown in Chapter 5, Section 5.3. With a higher number of frameworks we could try to remove noise from the event log and then execute the process discovery algorithm, but with our input sample we could not define what was noisy behavior, as shown in Table 18, Chapter 5.

6.2. Limitations and Future Work

There are some improvements that can be implemented for the Reuse Miner approach. Reuse Miner is composed of three main steps: I) the event log mining; II) the process

mining; and III) the RDL mining. Currently, each step is performed in a separated component and we could not automate the process because the use of ProM framework demands user interaction. The framework provides a command line feature and it also encourages developers to implement these features when creating and integrating new algorithms, but it is not a mandatory feature. In our case, the plugin that converted the causal-net to XPDL could only be used through the user interface. In this context, we identified two possible improvements for the future:

1. Evaluate the use of a different plugin to generate the BPMN model. There is a new process discovery algorithm in development phase that uses BPMN as its representational language, the BPMN Miner (CONFORTI et al., 2015). It is still not available in ProM's current release, but it could simplify Reuse Miner's process discovery.
2. Another improvement in the process discovery would be to have a tool that automates the repetition of the discovery of process and sub-processes. Depending on the input repositories and the number of sub-processes found, the number of event logs generated by Reuse Miner is high and the task of mining the instantiation process, separated in smaller recipes, demands a lot of interactions with the ProM framework to obtain the complete instantiation process. Currently, a ProM framework extension to the RapidMiner[5] tool, a workflow-based tool, is available for supporting the repeatability of the process discovery phase. However, not all plugins used in this dissertation were available in the ProM

[5] ProM for Rapid Miner: http://www.processmining.org/rapidminer/start

extension. Since the code for the algorithms is open-source, one alternative would be to integrate them into the RapidMiner extension, which was not implemented due to time constraints.

References

AALST, W. M. P. van der, 2011. **Process Mining: Discovery, Conformance and Enhancement of Business Processes**, 1st edition, Springer Publishing Company, Incorporated, 2011.

AALST, W. M. P. van der, 2013. **Business Process Management: A Comprehensive Survey**. ISRN Software Engineering, vol. 2013, pp. 1–37, 2013.

AALST, W. M. P. van der, ADRIANSYAH, A., DONGEN, B. F. van, 2011, "Causal Nets: A Modeling Language Tailored towards Process Discovery". In: **Proceedings of the 22nd International Conference - CONCUR 2011**, pp. 28–42, Aachen, Germany, September 2011.

AALST, W. M. P. van der, ADRIANSYAH, A., MEDEIROS, A. K. A. de, 2012. "Process Mining Manifesto". In: **Daniel, F., Barkaoui, K., Dustdar, S. (eds), BPM 2011 Workshops**, vol. 99, pp. 169–94. Springer, Heidelberg, 2012.

AGRAWAL, R., D. GUNOPULOS, and F. LEYMANN, 1998, "Mining Process Models from Workflow Logs". In: **Proceedings of the 6th International Conference on Extending Database Technology: Advances in Database Technology**, pp. 469–83, Valencia, Spain, March 1998.

XIE, T., and J. PEI. 2006. "MAPO: Mining API Usages from Open Source Repositories". In: **Proceedings of the 2006 International Workshop on Mining Software Repositories**, pp. 54–57. New York, NY, USA, 2006.

BLOCH, J. 2008. **Effective Java**. 2nd ed. Addison-Wesley.

BRAGA, R.T.V., MASIERO, P.C. 2003. "Building a Wizard for Framework Instantiation Based on a Pattern Language". In: **9th International Conference on Object-Oriented Information Systems**, pp. 95–106, Springer Berlin Heidelberg, September 2003.

BRAGA, R.T.V., MASIERO, P.C.. 2004. "Finding Framework Hot Spots in Pattern Languages". **Journal of Object Technology,** vol.3, no.1, pp. 123–142, Jan 2004.

BRUCH, M., MEZINI, M., MONPERRUS, M. 2010. “Mining Subclassing Directives to Improve Framework Reuse”. In: **Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR'2010)**, pp. 141–150, Cape Town, South Africa, May 2010.

BRUCH, M., MONPERRUS, M., MEZINI, M. 2009. “Learning from Examples to Improve Code Completion Systems”. In: **Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering**, pp. 213–222, New York, NY, USA, August 2009.

CONFORTI, R., DUMAS, M., GARCÍA-BAÑUELOS, *et al*., 1995. “BPMN Miner: Automated Discovery of BPMN Process Models with Hierarchical Structure”. Available in: <http://eprints.qut.edu.au/83646> Accessed on May 31 2015.

COOK, J. E., WOLF, A. L. 1995. “Automating Process Discovery Through Event-Data Analysis”. In: **Proceedings of the 17th International Conference on Software Engineering**, pp. 73-82, Seattle, Washington, USA, April 1995.

COOK, J. E., WOLF., A. L., 1998, “Discovering Models of Software Processes from Event-Based Data”. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, vol. 7, no. 3, pp. 215–49, July 1998.

DATTA, A., 1998, “Automating the Discovery of As-Is Business Process Models: Probabilistic and Algorithmic Approaches”. **Information Systems Research**, vol. 9, no. 3, pp. 275–301, March 1998.

DONGEN, B. F. van, 2005, “A Meta Model for Process Mining Data”. In: **Proceedings of the Open Interop Workshop on Enterprise Modelling and Ontologies for Interoperability, Co-located with CAiSE'05 Conference**, pp. 309–20, Porto, Portugal, June 2005.

DONGEN, B. F. van, MEDEIROS, A .K. A. de , WEN, L., 2009. “Process Mining: Overview and Outlook of Petri Net Discovery Algorithms”. In: **Jensen, K., Aalst, W. M. P. van der (eds), Transactions on Petri Nets and Other Models of Concurrency II**, Springer Berlin Heidelberg, 2009.

ESTUBLIER, J. 2006. “Software Are Processes Too”. In: **Unifying the Software Process Spectrum**, pp. 25–34. Springer Berlin Heidelberg, 2006.

FAYAD, M., Schmidt, D. C. 1997, “Object-Oriented Application Frameworks”. **Communications of the ACM**, vol.40, n.10, pp. 32–38, October 1997.

FRAKES, W.B., KANG, K, 2005, “Software Reuse Research: Status and Future”. **IEEE Transactions on Software Engineering**, vol. 31, no. 7, pp. 529–536, July 2005.

GANGOPADHYAY, D., MITRA, S. 1995. “Understanding Frameworks by Exploration of Exemplars”. In: **Proceedings of the 7th International Conference on Computer-Aided Software Engineering**, pp. 90–99. Toronto, Ontario, 1995.

GEF, 2002. “Eclipse GEF (Graphical Editing Framework)". Available in: <http://www.eclipse.org/gef/>. Accessed on May 31 2015.

GOMES, T. L., OLIVEIRA, T. C., COWAN, D., *et al.*, 2014. “Mining Reuse Processes”. In: **Proceedings of the XVII Ibero-American Conference on Software Enginering.**, Púcon, Chile, 2014.

HEYDARNOORI, A., CZARNECKI, K., BARTOLOMEI, T. T. 2009. “Supporting Framework Use via Automatically Extracted Concept-Implementation Templates”. In: **ECOOP 2009 – Object-Oriented Programming**, pp. 344–368, Springer Berlin Heidelberg

HEYDARNOORI, A., CZARNECKI, K., BINDER, W. *et al.* 2012. “Two Studies of Framework-Usage Templates Extracted from Dynamic Traces.” **IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,** pp. 1464-1487, vol. 38., December 2012

jbpt. 2013. “BPT Jbpt - Business Process Technologies 4 Java”. Available in: <https://code.google.com/p/jbpt/>. Accessed on May 29 2015.

JOHNSON, R. 1992. “Documenting Frameworks Using Patterns”. In: **Object-Oriented Programming Systems, Languages, and Applications**, pp. 63–76, vol. 27, no. 10, October 2012.

JOHNSON, R., PEARSON, D., PINGALI, K. 1994. “The Program Structure Tree: Computing Control Regions in Linear Time”. In: **Proceedings of the ACM *SIGPLAN 1994 Conference on Programming Language Design and Implementation***, pp. 171–185, New York, NY, USA, 1994.

KIRK, D., ROPER, M., WOOD, M. 2007. “Identifying and Addressing Problems in Object-Oriented Framework Reuse”. **Empirical Software Engineering**, vol. 12, no. 3, pp. 243–274, June 2007.

LETHBRIDGE, T.C., SINGER, J., FORWARD, A. 2003. “How Software Engineers Use Documentation: The State of the Practice”, vol. 20, no. 6, pp. 35–39, November 2003.

MARKIEWICZ, M., LUCENA, C. J. P. 2000. “Understanding Object-Oriented Framework Engineering”. PUC. Available in: <ftp://ftp.inf.puc-rio.br/pub/docs/techreports/00_38_markiewicz.pdf>. Accessed on 31 May 2015.

MATTSSON, M. 1996. **Object-Oriented Frameworks: A Survey of Methodological Issues**, Department of Computer Science, Lund University, Lund Institute of Technology, Lund, Sweden, 1996.

MEDEIROS, A. K. A. de, 2006. **Genetic Process Mining**, Ph.D.Thesis, Technische Universiteit Eindhoven, Eindhoven, The Nederlands, 2006.

MICHAIL, A. 2001. “CodeWeb: Data Mining Library Reuse Pattern.” In: **Proceedings of the 23rd International Conference on Software Engineering**, pp. 827–28. Waikiki, Honolulu, Hawaii, 2011.

OLIVEIRA, T. C., ALENCAR, P., COWAN, D. 2011. “ReuseTool - An Extensible Tool Support for Object-Oriented Framework Reuse”. **Journal of Systems and Software,** vol. 84, no. 12, pp. 2234–52, December 2011.

OLIVEIRA, T.C., ALENCAR, P.S.C., LUCENA, C. J. P., *et al.,* 2007. “RDL: A Language for Framework Instantiation Representation”. **Journal of Systems and Software**, vol. 80, no. 11, pp. 1902–1929, November 2007.

OMG, 2011. “Business Process Model and Notation (BPMN)". Available in: <http://www.omg.org/spec/BPMN/2.0/>. Accessed on 29 May, 2015.

ORTIGOSA, A., CAMPO, M. 1999. “SmartBooks: A Step beyond Active-Cookbooks to Aid in Framework Instantiation”. In: **Proceedings of Technology of Object-Oriented Languages and Systems**, pp. 131–140, June 1999, Santa Barbara, CA, USA.

ORTIGOSA. 2000. “Using Incremental Planning to Foster Application Frameworks Reuse”. **International Journal of Software Engineering and Knowledge Engineering**, vol. 10, no. 4, pp. 433–448, 2000.

PONCIN, W., SEREBRENIK, A., BRAND, M. van den. 2011. “Process Mining Software Repositories”. In: **2011 15th European Conference on Software Maintenance and Reengineering (CSMR)**, pp. 5 –14, March 2011, Szeged, Hungary.

PREE, W. 1995. “Framework Development and Reuse Support”. **Visual Object-Oriented Programming, Concepts and Environments**, pp. 253–67, Manning - Prentice Hall, Greenwich, CT, USA, 1995.

SALVADOR, G., 2009, **Métodos Empíricos para Validação da Reuse Description Language em Instanciação de Frameworks**. Masters Dissertation, Pontifícia Universidade Católica do Rio Grande do Sul, Rio Grande do Sul, Brazil, 2009.

SILVA JUNIOR, L.L.N., OLIVEIRA, T.N., PLASTINO, A. *et al*. 2012. “Vertical Code Completion: Going Beyond the Current Ctrl+Space”. In: **Proceedings of the 2012 Sixth Brazilian Symposium on Software Components, Architectures and Reuse**, pp. 81–90, September 2012.

THUMMALAPENTA, S., XIE, T., 2008. “SpotWeb: Detecting Framework Hotspots and Coldspots via Mining Open Source Code on the Web”, In: **Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering**, pp. 327–36, L'Aquila, Italy, September 2008.

VANHATALO, J., VÖLZER, H., KOEHLER, J. 2008. “The Refined Process Structure Tree”. In **Proceedings of the 6th International Conference on Business Process Management**, pp. 100–115, 2008, Milan, Italy.

VANHATALO, J., VÖLZER, H., LEYMANN, F. 2007. “Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE

Decomposition". In: **Proceedings of the 5th International Conference on Service-Oriented Computing**, pp. 43–55, 2007, Vienna, Austria.

VERBEEK, H. M. W., BUIJS, J. C. A. M., DONGEN, B. F. Van *et al*. 2011. "XES, XESame, and ProM 6". In: **Soffer, P. Proper, E. (eds) Information Systems Evolution (CAiSE Forum 2010)**, vol. 72, pp. 60–75, Berlin, Springer, 2011.

WEERDT, J. de, BACKER, M. de, VANTHIENEN, J., BAESENS, B., 2012, "A Multi-Dimensional Quality Assessment of State-of-the-Art Process Discovery Algorithms Using Real-Life Event Logs". **Information Systems**, vol. 37, no. 7, pp. 654–676, November 2012.

WFMC. 2012. "XPDL - XML Process Definition Language". Available in <http://www.xpdl.org/standards/xpdl-2.2/XPDL%202.2%20(2012-08-30).pdf>. Accessed on May 29 2015.

XIE, T., and J. PEI. 2006. "MAPO: Mining API Usages from Open Source Repositories". In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, pp. 54–57. New York, NY, USA, May 2006.

ZHONG, H., XIE, T., ZHANG, L., *et al.*, 2009. "MAPO: Mining and Recommending API Usage Patterns". In" **Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming**, pp. 318–43. Genova, Italy, July 2009.

Appendix A – Graphiti RDL

This appendix shows the RDL process manually created for Graphiti framework based on the framework online help. This script was used as a reference for computing precision and recall in Chapter 5.

Table 21. Main recipe

```
1.  recipe Main {
2.      external_task ("Create an eclipse plugin");
3.      pack = new_package(FrameworkModel, "?");
4.
5.     // Call recipe
6.      AbstractDiagramTypeProvider();
7.
8.      external_task("Register the diagram in plugin.xml");
9.
10.     if ("Contribute to eclipse's property feature?") {
11.         class_extension("org.eclipse.graphiti.ui.platform.
12.                 AbstractPropertySectionFilter", "?");
13.         class_extension("org.eclipse.graphiti.ui.platform.
14.                 GFPropertySection", "?");
15.         external_task("Register extension point in plugin.xml");
16.     }
17.
18.     if ("Create image provider?") {
19.         class_extension("org.eclipse.graphiti.ui.platform.
20.                 AbstractImageProvider", "?");
21.         external_task("Register extension point in plugin.xml");
22.     }
    }
```

Table 22. AbstractDiagramTypeProvider Recipe

```
1.  recipe AbstractDiagramTypeProvider {
2.
3.      // Creates a default Feature Provider
4.      DefaultFeatureProvider();
5.
6.      // Creates a default tool behavior Provider
7.      class_extension("org.eclipse.graphiti.tb.
8.              DefaultToolBehaviorProvider", "?");
9.
```

```
10.     // Creates a diagram type
11.     class_extension("org.eclipse.graphiti.dt.
12.                     AbstractDiagramTypeProvider", "?");
13.  }
```

Table 23. DefaultFeatureProvider recipe

```
 1.  recipe DefaultFeatureProvider {
 2.
 3.     loop ("Create layout feature?") {
 4.     // Layout features mainly supports the recalculation of positions
 5.     // and sizes inside the pictogram model. The corresponding add
 6.     //feature must know this class
 7.        class_extension("org.eclipse.graphiti.features.impl.
 8.                        AbstractLayoutFeature", "?");
 9.     }
10.
11.     loop ("Create new add feature?") {
12.     // In the terms of  Graphiti “add” means to add a graphical
13.     // representation of an existing domain model object (= business
14.     // object) to the diagram. Here, we need to use one of the
15.     // implementations of the interface IAddFeature
16.        class_extension("org.eclipse.graphiti.features.impl.
17.                         AbstractAddShapeFeature", "?");
18.     }
19.
20.     loop ("New create feature?") {
21.     // Create a new graphical representation for a business model. For
22.     // adding the graphical representation, the existing add feature
23.     // should be used
24.        class_extension("org.eclipse.graphiti.features.impl.
25.                         AbstractCreateFeature", "?");
26.     }
27.
28.     loop ("Create new connection feature?") {
29.     // In the terms of  Graphiti “add” means to add a graphical
30.     // representation of an existing domain model object (= business
31.     // object) to the diagram. Here, we need to use one of the
32.     // implementations of the interface IAddFeature
33.        class_extension("org.eclipse.graphiti.features.impl.
34.                         AbstractAddShapeFeature", "?");
35.        class_extension("org.eclipse.graphiti.features.impl.
36.                         AbstractCreateConnectionFeature", "?");
37.        class_extension("org.eclipse.graphiti.features.impl.
38.                         DefaultReconnectionFeature", "?");
39.     }
40.
41.     loop ("Create a new update feature?") {
42.     // Create a new graphical representation for a business model. For
43.     // adding the graphical representation, the existing add feature
44.     // should be used
```

```
45.         class_extension("org.eclipse.graphiti.features.impl.
46.                         AbstractUpdateFeature", "?");
47.     }
48.
49.     // In most cases it is not necessary to provide an own implementation
50.     // of the remove feature.
51.     loop ("Create new custom implementation for the remove feature?") {
52.       class_extension("org.eclipse.graphiti.features.impl.
53.                         DefaultRemoveFeature", "?");
54.     }
55.
56.     // In most cases, the graphics framework has enough knowledge to
57.     // provide a good default implementation for the delete feature
58.     loop ("Create new custom implementation for the delete feature?") {
59.       class_extension("org.eclipse.graphiti.features.impl.
60.                         DefaultDeleteFeature", "?");
61.     }
62.
63.     // The framework provides a move feature by default and the reuser
64.     // should decide if it needs customization
65.     loop ("Create a custom move feature?") {
66.       class_extension("org.eclipse.graphiti.features.impl.
67.                         DefaultMoveShapeFeature", "?");
68.     }
69.
70.     // The framework provides a move feature by default and the reuser
71.     // should decide if it needs customization
72.     loop ("Create a custom resize feature?") {
73.       class_extension("org.eclipse.graphiti.features.impl.
74.                         DefaultResizeShapeFeature", "?");
75.     }
76.
77.     loop ("Create custom feature?") {
78.       class_extension("org.eclipse.graphiti.features.impl.
79.                         AbstractCustomFeature", "?");
80.     }
81.
82.     if ("Enable direct editing?") {
83.       class_extension("org.eclipse.graphiti.features.impl.
84.                         AbstractDirectEditingFeature", "?");
85.     }
86.
87.     if ("Create copy feature?") {
88.       class_extension("org.eclipse.graphiti.ui.features.
89.                         AbstractCopyFeature", "?");
90.     }
91.
92.     if ("Create paste feature?") {
93.       class_extension("org.eclipse.graphiti.ui.features.
94.                         AbstractPasteFeature", "?");
95.     }
96.
97.     if ("Create drill down feature?") {
```

```
98.         // This feature should be returned in the default feature
99.         // provider in the getCustomFeatures method
100.          class_extension("org.eclipse.graphiti.ui.features.
101.                        AbstractDrillDownFeature", "?");
102.
103.      }
104.
105.      // Creates the default feature provider for the existing editor
106.      class_extension("org.eclipse.graphiti.ui.features.
107.                    DefaultFeatureProvider", "?");
108.  }
109.
```

Appendix B – GEF RDL

This appendix shows the RDL process manually created for GEF framework based on five framework applications. This script was used as a reference for computing precision and recall in Chapter 5.

Table 24. EditPartFactory recipe

```
1.  recipe EditPartFactory {
2.
3.      if ("Create edit part factory for tree edit part?") {
4.          loop ("create AbstractTreeEditPart?") {
5.              class_extension("org.eclipse.gef.editparts.
6.                      AbstractTreeEditPart", "?");
7.          }
8.      } else {
9.          if ("Create connection?") {
10.             class_extension("org.eclipse.gef.editparts.
11.                     AbstractConnectionEditPart", "?");
12.         }
13.
14.         loop ("Create graphical edit part?") {
15.             graphicalEditPart = class_extension("org.eclipse.
16.                 gef.editparts.AbstractGraphicalEditPart", "?");
17.
18.             if ("implement NodeEditPart?") {
19.                 new_interface_realization(graphicalEditPart,
20.                             "org.eclipse.gef.NodeEditPart");
21.             }
22.         }
23.     }
24.     class_extension("org.eclipse.gef.EditPartFactory", "?");
25. }
```

Table 25. AbstractConnectionEditPart recipe

```
1.  recipe AbstractConnectionEditPart {
2.      class_extension("org.eclipse.gef.editpolicies.
3.              ConnectionEditPolicy", "?");
4.      class_extension("org.eclipse.gef.editpolicies.
5.              ConnectionEndpointEditPolicy", "?");
6.
7.      if ("create BendpointEditPolicy?") {
8.          class_extension("org.eclipse.gef.editpolicies.
```

```
                    BendpointEditPolicy", "?");
    }

    if ("create new command?") {
        class_extension("org.eclipse.gef.commands.Command", "?");
    }

    class_extension("AbstractConnectionEditPart", "?");
}
```

Table 26. AbstractGraphicalEditPart Recipe

```
recipe AbstractGraphicalEditPart {

    if ("Create SelectionEditPolicy") {
        class_extension("org.eclipse.gef.editpolicies.
                SelectionEditPolicy", "?");
    }

    if ("Create ContainerEditPolicy & LayoutEditPolicy") {
        class_extension("org.eclipse.gef.editpolicies.
                ContainerEditPolicy", "?");
        class_extension("org.eclipse.gef.editpolicies.
                LayoutEditPolicy", "?");
    }

    if ("Create FlowLayoutEditPolicy") {
        class_extension("org.eclipse.gef.editpolicies.
                FlowLayoutEditPolicy", "?");
    }

    if ("Create XYLayoutEditPolicy?") {
        class_extension("org.eclipse.gef.editpolicies.
                XYLayoutEditPolicy", "?");
    }

    if ("Create GraphicalEditPolicy") {
        class_extension("org.eclipse.gef.editpolicies.
                GraphicalEditPolicy", "?");
    }

    if ("Create ScrollableSelectionFeedbackEditPolicy") {
        class_extension("org.eclipse.gef.editpolicies.
                ScrollableSelectionFeedbackEditPolicy", "?");
    }

    if ("Create ComponentEditPolicy) {
        class_extension("org.eclipse.gef.editpolicies.
                ComponentEditPolicy", "?");

        if ("Create GraphicalNodeEditPolicy?") {
```

```
40.             class_extension("org.eclipse.gef.editpolicies.
41.                         GraphicalNodeEditPolicy", "?");
42.         }
43.     }
44.
45.     if ("Create DirectEditPolicy?") {
46.         class_extension("org.eclipse.gef.editpolicies.
47.                     DirectEditPolicy", "?");
48.     }
49.
50.     if ("Add direct editing funcionality?") {
51.         class_extension("CellEditorLocator", "?");
52.         class_extension("org.eclipse.gef.tools.DirectEditManager", "?");
53.     }
54.
55.     loop("Create new command") {
56.         class_extension("org.eclipse.gef.commands.Command", "?");
57.     }
58.
59.     class_extension("org.eclipse.gef.editparts.
60.                 AbstractGraphicalEditPart", "?");
61. }
```