Graph-based Pattern Matching and Discovery for Process-centric Service Architecture Design and Integration

Veronica Gacitua-Decar

B.Eng. (UTFSM) 2003
M.Sc. (UTFSM) 2004

A Dissertation submitted in fulfilment
of the requirements for the award of

Doctor of Philosophy (Ph.D.)

to the

DUBLIN CITY UNIVERSITY
FACULTY OF ENGINEERING AND COMPUTING
SCHOOL OF COMPUTING

Advisor: Dr. Claus Pahl

June, 2010

Examiners :

Dr. Markus Helfert	-	School of Computing, Dublin City University
Prof. Dr. Reiko Heckel	-	Department of Computer Science, University of Leicester

Acknowledgments

Looking back upon the path this journey has taken, I feel very fortunate to have had the opportunity to learn from a life enriching experience and to share moments with wonderful people. I was lucky to have explored not only a new field of knowledge, but also a new language and different cultures.

I would like to express my sincere gratitude to those who made this thesis possible and made the time during this PhD even more valuable.

I am deeply thankful to my advisor and colleagues in DCU.

To Claus my sincere gratitude for always having an endless and constant patience, willingness to support me and give me all the freedom to develop my own ideas.

To my colleagues in the Software and System Engineering Group, thank you so much for sharing all those brilliant and cloudy moments, full of laughs, conversations, discussions and hard work. Ronan and Mark thanks for receiving me with open arms and showing me the exciting Irish life. Mark, you always give me optimist and strength to continue. Thanks Wong for always having very intelligent and sharp opinions regarding our work but also about politics and life matters. Kosala, Aakash and Javed, thanks for helping me through this journey and opening my knowledge to other cultures and religions. I learned many things from all of you.

To my bay-mates and colleagues in Lero: Declan, Haiying, Oisin, Michele, Darren, Paul, Michal, Murat, Aarthy, Yalemisew, who supported me in a number of ways, from positive energy during breaks, to excellent morning coffees that made me able to continue during the day, to discussions during the preparation of presentations, to excellent asian food! Thanks all for your great camaraderie.

I am immensely grateful to Laly and Alberto who help me as family during my days in Dublin. To Paola for her constant and generous friendship that I feel will last my lifetime. To Elisabeth and Jeff who helped me through the last months of writing this thesis with a wonderful friendship.

I am forever indebted to my family – mama, papa y hermano – for their unconditional love and constant care although I was physically far from them. They taught me what is important in life, including the strong sense of perseverance, that made this work possible. Gracias por todo.

I owe my deepest gratitude to my beloved Ivan who was day by day, hour by hour, second by second giving me his time, love and soul. Thank you for making me a better person every day.

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy is entirely my own work, that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed :
(Veronica Gacitua-Decar)

Student ID : 56124821

Date : June 26, 2010

Graph-based Pattern Matching and Discovery for Process-centric Service Architecture Design and Integration

Abstract: Process automation and applications integration initiatives are often complex and involve significant resources in large organisations. The increasing adoption of service-based architectures to solve integration problems and the widely accepted practice of utilising patterns as a medium to reuse design knowledge motivated the definition of this work. In this work a pattern-based framework and techniques providing automation and structure to address the process and application integration problem are proposed. The framework is a layered architecture providing modelling and traceability support to different abstraction layers of the integration problem. To define new services - building blocks of the integration solution - the framework includes techniques to identify process patterns in concrete process models. Graphs and graph morphisms provide a formal basis to represent patterns and their relation to models. A family of graph-based algorithms support automation during matching and discovery of patterns in layered process service models. The framework and techniques are demonstrated in a case study. The algorithms implementing the pattern matching and discovery techniques are investigated through a set of experiments from an empirical evaluation. Observations from conducted interviews to practitioners provide suggestions to enhance the proposed techniques and direct future work regarding analysis tasks in process integration initiatives.

Keywords: service-oriented architecture, enterprise application integration, business process management, process pattern, architectural pattern, pattern matching, pattern discovery, graph matching, frequent subgraph discovery.

Contents

List of Figures

List of Tables

List of Publications

- Gacitua, V. and C. Pahl (2007). *Business Process-driven Service Architecture Reuse*. ERCIM News. Special Theme: Service Computing. 70: 49-50.
- Gacitua-Decar, V. and C. Pahl (2008). *Business model driven design of service architectures for Enterprise Applications Integration: A pattern-based approach*. DCSOFT'08: Proceedings of the Doctoral Consortium on Software and Data Technologies, INSTICC.
- Gacitua-Decar, V. and C. Pahl (2008). *Business model driven Service Architecture Design for Enterprise Application Integration*. ICBIIT'08: Proceedings of the International Conference on Business Innovation and Information Technology, Logos Verlag.
- Gacitua-Decar, V. and C. Pahl (2008). *Pattern-based business-driven analysis and design of service architectures*. ICSOFT (SE/MUSE/GSDCA)'08: Proceedings of the Third International Conference on Software and Data Technologies, INSTICC.
- Gacitua-Decar, V. and C. Pahl (2008). *Service Architecture Design for E-Businesses: A Pattern-Based Approach*. EC-Web'08: Proceedings of the 9th International Conference on Electronic Commerce and Web Technologies, Springer.
- Gacitua-Decar, V. and C. Pahl (2008). *Towards Pattern-Based Service Identification*. WEWST'08: Proceedings of the 3rd Workshop on Emerging Web Services Technology.
- Gacitua-Decar, V. and C. Pahl (2009). *Automatic Business Process Pattern Matching for Enterprise Services Design*. SERVICES-2'09: Proceedings of the 2009 IEEE World Conference on Services - II, IEEE.
- Gacitua-Decar, V. and C. Pahl (2009). *Ontology-based Patterns for the Integration of Business Processes and Enterprise Application Architectures*. In Book: *Semantic Enterprise Application Integration*, G. Mentzas, T. Bouras, P. Gouvas and A. Friesen (Eds.), IGI Publishers, Ltd.
- Gacitua-Decar, V. and C. Pahl (2009). *Towards Reuse of Business Processes Patterns to Design Services*. In: Emerging Web Services Technology. W. Binder and S. Dustdar (Eds.), Springer - Birkhauser. III: 15-36.
- Garcia-Gonzalez, J. P., G. Gacitua-Decar, et al. (2009). *A Service Architecture Solution for Mobile Enterprise Resources: A Case Study in the Banking Industry*. Emerging Web Services Technology. W. Binder and S. Dustdar (Eds.), Springer - Birkhauser. Volume III: 143-155.
- Garcia-Gonzalez, J. P., V. Gacitua-Decar, et al. (2009). *Service Registry: A Key Piece to Enhance Reuse in SOA*. The Architecture Journal, Microsoft. **21**:29-34.
- Pahl, C., Y. Zhu, et al. (2009). *A Template-driven Approach for Maintainable Service-oriented Information Systems Integration*. International Journal of Software Engineering and Knowledge Engineering, World Scientific. **19**(7):889-912.

List of Abbreviations

A	: Accuracy
AAL	: Application Architecture Layer
AG	: Attributed Graph
ALMA	: Architecture-level modifiability analysis
ATG(M/P)	: Attributed Type/typed Graph (Model/Pattern)
Constr	: Constraints
BAIL	: Business-Applications Intermediate Layer
BML	: Business Modelling Layer
BPM	: Business Process Management
BPMN	: Business Process Modelling Notation
CP-E-PM	: Complete/Partial and Exact Pattern Matching
CP-I-PM	: Complete/Partial and Inexact Pattern Matching
CP-I-Attr-PM	: CP-I-PM with relaxed Attribute Matching
CP-I-Type-PM	: CP-I-PM with relaxed Type Matching
CP-I-Strc-PM	: CP-I-PM with relaxed Structural preserving mapping
dis	: Dissimilarity
EAI	: Enterprise Application Integration
EBPP	: Electronic Bill Presentment and Payment
ecm	: Exact and complete mapping
epm	: Exact and partial mapping
H-PM	: Hierarchical Pattern Matching
IEC	: International Electrotechnical Commission
ISO	: International Organization for Standardization
LABAS	: Layered Architecture for Business, Applications and Services
LCGH	: Locally Constrained Graph Homomorphism
LKB	: Lexical Knowledge Base
LM	: Loan Management
mA	: Match Accuracy
maxOutDeg	: Maximum Out Degree
$N_G(u)$	: Neighbourhood of u in G
NRA	: National Revenue Agency
OMG	: Object Management Group
P	: Precision
PI	: Pattern instance
R	: Recall
rec_M	: Set of recorded graph models
SAL	: Service Architecture Layer
sim	: Similarity
SOA	: Service Oriented Architecture
TG	: Type/typed Graph
UML	: Unified Modelling Language
WS-BPEL	: Web Services Business Process Execution Language

Chapter 1

Introduction

Contents

1.1 Motivation

Software applications in organisations are built or acquired to provide specialised functionality required to support business processes. If new activities and applications are created and integrated into existing business processes and infrastructures, new architecture and information requirements have to be satisfied. Enterprise Application Integration (EAI) and Business Process Management (BPM) aim to link separate applications into an integrated system driven by business models and the goals they implement, and to improve productivity, product quality and operations of an enterprise, respectively. In recent years, Service-Oriented Architecture (SOA) appeared as an architectural approach that has the potential to bridge the gap between business and technology and to improve the reuse of existing applications and the interoperability with new ones. Software services are the building blocks of SOA. They can be composed to provide more complex functionality and to automate business processes. However, to make the design of a service actually reusable across processes and organisations is a challenging, manual and often iterative task. Together with the challenge of designing adequate services, service architectures have to be coherently maintained and closely aligned with business processes, although applications are created without a structured architectural design and processes are continuously changing due to new regulations, organisational mergers or operational redesigns.

The development of integrated enterprise- and process-wide application architectures is a continuous process. To improve the process and overall quality, the experience of analysts, architects and developers should be captured and reused. Abstraction is a central driver in software engineering approaches and – at the business level – the reuse of successful business designs is equally important. Abstraction and knowledge representation are principles that can assist in dealing with the challenges of SOA design and integration. On the other hand, the size and complexity of models involved with enterprise SOA make the tasks of analysts and architects extraordinarily time-consuming and susceptible to numerous errors. Any automated assistance during the analysis and design of SOA can benefit the overall goal of processes and applications integration.

1.2 Overview of Problems with the State of the Art

SOA methodologies have advanced the way services and their organisations are designed and developed. However, there are still problems to be addressed such as providing adequate integration between modelling views involved in enterprise processes and application integration problems, providing guidelines and tool support to automate steps during the design of services that have an adequate granularity and satisfy the constraints imposed by existing applications and the processes they integrate, and providing the means to reuse knowledge through documented architecture and process abstractions. In this work, the concept of pattern, as a medium to capture process-centric architectural abstractions, is a central idea used to assist the design of new services. Only a few SOA development methodologies have explicit support to work with patterns. Most of them focus on providing facilities to document new patterns, store them and then helping to apply those patterns in working models. Moreover, most patterns are based at the software level. For process-centric service architectures designed to solve enterprise processes and applications integration problems, process level patterns are fundamental. Existing tools facilitating pattern storage and utilisation are important; however, a main problem is pattern identification. The experience of analysts and architects in the context of process-centric systems is embedded in large and complex layered process-centric descriptions. Hand-crafted approaches to identify process patterns in large and complex process models are subject to errors and excessive time consumption, making them difficult to adopt in practice. Process pattern identification is a step preceding pattern application. Automating pattern identification in layered process-centric descriptions is the challenge addressed in this work. Two main scenarios are addressed. One focuses on finding instances of known patterns (pattern matching), and the other one focuses on discovering unknown patterns (pattern discovery). Pattern

matching and discovery in process and architecture models can be abstracted to the graph matching and frequent subgraph discovery problems. These problems have been addressed in different fields but only recently have been associated to problems involving process-centric models. Process models contain rich descriptions of their elements and capture the idea of behaviour; therefore, graphs capturing these models have special characteristics. Graph matching and frequent subgraph discovery in process models go beyond techniques provided in other fields, requiring specific solutions that take into account not only the graph structure but also the semantics of process element descriptions and the behaviour that process models attempt to capture.

1.3 Contribution

1.3.1 Hypothesis and Research Questions

This work aims to contribute techniques to automate and support the reuse of process-centric architecture and model abstractions to improve the current approaches for enterprise SOA design and integration.

Hypothesis. A pattern-based framework and techniques for layered service process models can benefit

- maintainability,
- functional suitability and compliance, and
- traceability

of service-based systems for enterprise processes and applications integration.

In particular, process pattern matching and discovery techniques can be automated while addressing requirements related to effectiveness and efficiency, involving

- feedback to end users on the matching and discovery results,
- the ability of the techniques to accommodate to variations in their inputs by providing flexible outputs such as partial or inexact results,
- if the results are not the intended results the effort to modify them should be less than the effort to perform the task manually, and
- the processing time for automatically matching or discovering patterns should be significantly improved compared with a hand-crafted approach.

Research Questions. According to [Shaw 2002], an adequate research in software engineering would depend on the type of questions defined for the investigation. For

methods or means of development, the question How do we do/create or automate the method? is central. For analysis methods, How can I evaluate the quality/-correctness of the method? For the design, evaluation or analysis of a particular technique/product, How does the technique perform in relation to a particular measure? For generalisation and characterisation, What are the important characteristics and varieties of the technique? How are they related? For feasibility, Is it possible to accomplish the intended results? Considering these recommendations and the hypothesis described above, basic research questions were derived and include

- How to organise and maintain related models from different layers involved in enterprise processes and applications integration?
- How can design knowledge in the form of patterns be integrated to this organisation?
- Can an organised framework and pattern-based techniques benefit maintainability, functional suitability/compliance and traceability?
- Which properties of processes are critical in identifying patterns?
- How can these properties be represented?
- How can known patterns be identified in processes? Can this be automated?
- Can unknown patterns be identified in processes? Can this be automated?
- Can known and unknown patterns be effectively and efficiently identified?
- How can effectiveness and efficiency of pattern identification techniques be evaluated?

1.3.2 Proposal

Based on the hypothesis and research questions this work proposes a pattern-based framework to structure the integration problem in a layered architecture involving business, service and application modelling layers supported by pattern-based techniques to assist analysis and design tasks.

The framework focuses on the support for modelling and integration through explicit traceability and a set of pattern-based techniques. A dedicated profile and traceability model are proposed to support an integrated view of models during SOA design and integration. Among the pattern-based techniques considered in the proposed framework, two techniques for automated pattern matching and discovery are proposed and investigated in details. The techniques are based on a family of algorithms for graphs. A family of graph matching algorithms is the basis of the pattern matching technique. The graph matching algorithms can identify exact and inexact, as well as complete and partial matches. Also, a hierarchical graph matching algorithm is introduced. An algorithm for frequent subgraph discovery is the basis of the proposed pattern discovery technique. It recursively uses the algorithm for

exact and partial graph matching.

The framework is open to incorporate advances in any of the introduced pattern-based techniques. The pattern matching and discovery techniques have addressed the complexity of real process-centric models. They have the potential to be further extended to scenarios where the dynamics of model modifications becomes more critical.

1.4 Research Approach and Evaluation Methodology

Software engineering research is concerned with improving the ability to systematically and predictably analyse and develop software that satisfies complex quality-centric requirements – often expressed in *illities* – and that must evolve during its lifetime. Software engineering results often take the form of methods for development and analysis, they can be new or improved models or theories, and they may be established qualitatively, through empirical study of software systems, through empirical study of the software development process, or through formal analysis [SEI 2010].

Two main streams of research can be identified from behavioural science and design science [Hevner 2004]. While behavioural science addresses research through the development and justification of theories that explain or predict phenomena related to the identified requirements, design science addresses research through the building and evaluation of artifacts designed to meet the identified requirements. The goal of behavioural science research is truth. The goal of design-science research is utility. The research approach followed in this work is based in principles of design science.

A number of guidelines for conducting and evaluating design-science are provided in [Hevner 2004], [Hevner 2010]. The relation between the development of this thesis and these guidelines can be summarised as follow (Table 1.1 indicates the relations between guidelines and chapters in this thesis):

- Guideline 1: Design an artifact in the form of a construct, a model, a method, or an instantiation – An architectural and modelling framework provides a general model to capture the different perspectives involved in SOA design and integration, i.e., business model, service architecture and application architecture. The framework also provides explicit traceability between elements of its model. Constructs to capture model abstractions (processes pattern) and a method (supported by techniques) to exploit their use for service identification are provided. Instantiation of the framework in a concrete chain tool facilitates its evaluation.

- Guideline 2: Develop technology-based solutions to important and relevant business problems – Enterprise services design and application integration is a relevant and current problem in organisations. Automating activities during services development and targeting pattern identification can be beneficial to reduce high costs of maintenance, compliance with regulations and poor reuse of design knowledge.
- Guideline 3: The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods. The evaluation method involves an scenario-based evaluation with case studies for the proposed framework. An experimental method including a prototype chain tool is utilised to evaluate the proposed algorithms for (process) pattern matching and discovery. Adequate metrics are defined for each type of evaluation. Interviews to practitioners in the field provided feedback to the proposed artifacts.
- Guideline 4: Provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies – Descriptions of foundational and related works are provided in the next Chapter 2. It gives an overview of the contribution (and relation) of this thesis in regard with SOA design and integration, and specifically, with the identification (matching and discovery) of process patterns to aid the definition of new services.
- Guideline 5: Apply rigorous methods in both the construction and evaluation of the design artifact – The main stages of the research, that involve the construction of the proposed framework and techniques, consider activities of requirements elicitation, development of the framework and pattern identification techniques. The evaluation method was already referred in Guideline 3 and it is discussed in more details (together with the requirements elicitation and development approach) in the following paragraphs.
- Guideline 6: Utilise available means to search for an effective artifact while satisfying laws in the problem environment – A stage of requirement elicitation and definition of initial case studies gave a sense of the available means to provide an effective framework in the context of enterprise SOA design and integration.
- Guideline 7: The research must be presented effectively both to technology-oriented as well as management-oriented audiences – Communication of the research in the form of publications is targeted to an academic and technology-oriented audience. A complementary assessment involving interviews to professionals in industry targeted not only an industry and technology-oriented audience but a management-oriented audience.

Table 1.1: Design science guidelines [Hevner 2010] and chapters in this thesis.

Guideline	Chapter (Ch)									
	Ch1	Ch2	Ch3	Ch4	Ch5	Ch6	Ch7	Ch8	Ch9	Ch10
Guideline 1	-	-	✓	-	✓	✓	-	-	-	-
Guideline 2	-	✓	-	-	-	-	-	-	-	-
Guideline 3	-	-	-	-	-	-	✓	✓	✓	-
Guideline 4	-	✓	-	-	-	-	-	-	-	✓
Guideline 5	-	-	-	-	-	-	✓	✓	✓	-
Guideline 6	-	-	-	-	-	-	✓	✓	✓	-
Guideline 7[1]	-	-	-	-	-	-	-	-	✓	-

Requirement Elicitation and Development Approach. Preliminary case studies were utilised to elicit requirements for the framework and pattern matching and discovery techniques. A goal-question-metric method was used to establish the metrics of interest, in this case focused on maintainability, functional suitability/compliance and traceability. The scenarios of the case study involved processes in the e-commerce and financial domains that include human interaction in processes and automated inter-organisational processes.

The modelling aspects of the framework were iteratively developed and focused on an effective use among end users. Modelling constructs were initially proposed; however, finally widely spread definitions coming from standardisation bodies were adopted. The purpose of this decision was improved usability. This implied the adoption of the abstract syntax for process models, service architectures and application architectures proposed from an external body from industry and their adaption to the proposed framework. These were developed using a profile. The core and distinguishing characteristics developed for the framework were related to the integration of models, traceability and pattern-based support.

The development of the pattern matching and discovery techniques was based on the creation of a family of graph-based algorithms. First, process-centric descriptions and patterns were defined as a special type of graphs. Data structures facilitating graph manipulation were used (in this case, matrix- and vector-based structures). Subsequently, the graph-based algorithms were developed in a iterative fashion. Initial pseudo-codes were refined before implementation. Executable versions of the algorithms were developed and critical circumstances tested, including termination. Errors encountered during the development phase were studied and root causes addressed. Similar to the general framework, a goal-question-metric method was used to establish the metrics of interest to assess the results obtained by the algorithms, and finally the techniques. The metrics attempt to measure the effectiveness and efficiency of the techniques and they include flexibility to identify partial and inexact results, accuracy of the results, time processing and the ability to provide feedback

to end users in a visual manner.

The overall proposal was evaluated considering three main parts:

- a scenario-based analysis method used for the framework,
- an empirical evaluation for pattern matching and discovery algorithms, and
- a complementary assessment based on interviews with practitioners.

Evaluation Approach. Enterprise SOA design and integration involve challenges at several levels of abstraction and relevance for different analysts and architects. In order to evaluate the work in this investigation, two main approaches have been adopted. The proposed framework is evaluated using two case studies and the guidelines of an scenario-based architecture analysis method. The scenario-based method is used to assess the benefits of the framework with regard to quality characteristics. Maintainability, functional suitability/compliance and traceability are the studied characteristics. In early design stages, these characteristics have a direct impact on the subsequent development stages and overall quality of the service-based system created for enterprise process and applications integration. An empirical evaluation involving a set of experiments is adopted to explore the effectiveness and efficiency of algorithms for pattern matching and discovery. A complementary assessment based on interviews with practitioners extends the evaluation process with a wider perspective that includes an 'industry' point of view. Based on these interviews, the ideas from a number of practitioners regarding model-based analysis and design techniques are explored.

Communication of the Research. Initial research involved with the proposed framework and techniques were presented to the academic community in the form of publications and reports. A limited group of practitioners was also communicated with during interview sessions that were a complementary part of this work's evaluation. For the research community, in addition to publications, the implementation of the techniques was made publicly available, offering the possibility to repeat the research or check correctness.

1.5 Organisation of the Thesis

The thesis document is summarised in Figure 1.1 and organised as follow.

- Chapter 2 provides a review of the literature describing contributions encompassing SOA methodologies, concepts enclosing ideas to represent and work with architecture and process model abstractions, and a number of techniques for service identification, pattern matching and discovery.

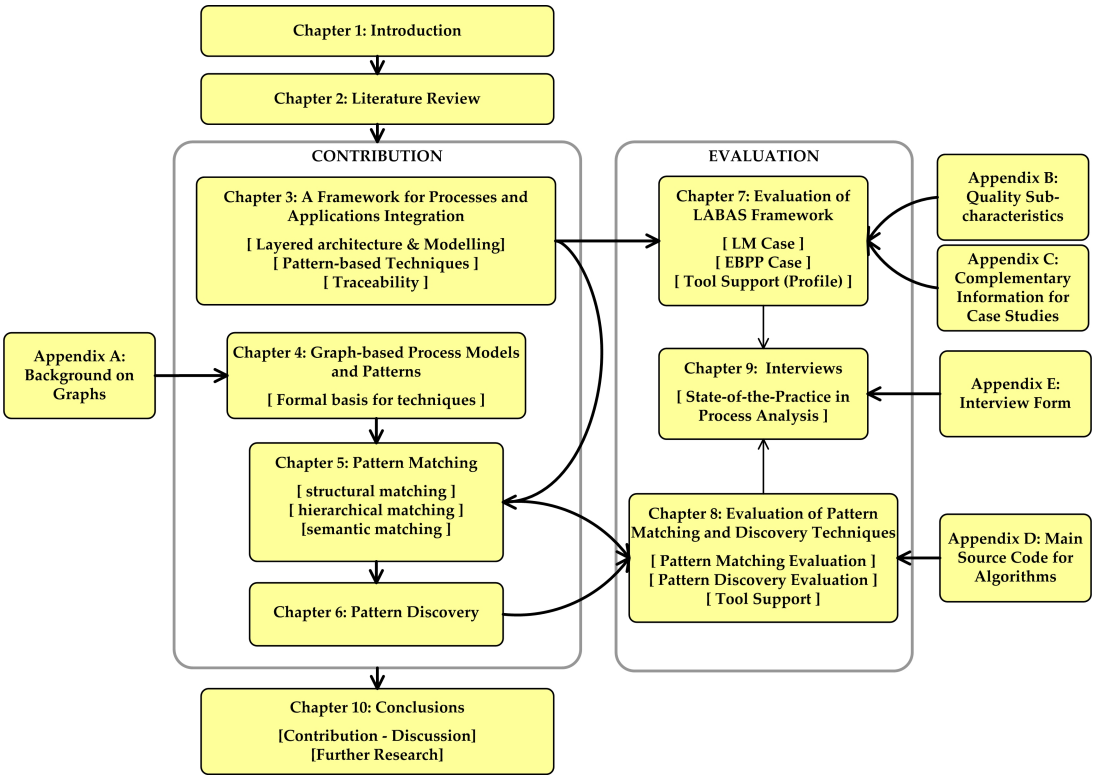


Figure 1.1: Organisation of this thesis.

- Chapter 3 describes the proposed pattern-based framework for SOA modelling and design. It explains the role of patterns, the abstraction layers, the traceability model and the techniques considered in the framework.
- Chapter 5 describes the proposed pattern matching techniques and underlying algorithms. The chapter is divided according to solutions for problems associated to structural and semantic aspects.
- Chapter 6 describes the proposed pattern discovery technique.
- The evaluation of the overall work and associated tool support is separated into three chapters. Modelling tool support and a case study-based assessment and discussion of the framework are explained in Chapter 7.
- The evaluation of the pattern matching and discovery techniques, together with tool support for implementing the algorithms, experimental setting and results visualisation are explained in Chapter 8.
- The results of a complementary assessment based on interviews with practitioners in industry are described in Chapter 9.
- Conclusions of this work and a description of possible further research are provided in Chapter 10.

Note that throughout the thesis, a number of sections are utilised to illustrate the use and benefits of the proposed techniques. These sections are a complement to the main structure of the thesis and they aim to facilitate the understanding of introduced concepts and applications.

Chapter 2

Literature Review

Contents

2.1 Overview

This chapter describes a review of the literature that is divided into five main sections. The next section introduces basic notions of enterprise application integration, business process management, service-oriented architectures and their relationships. It also briefly discusses the relevance of design knowledge reuse for enterprise processes and applications integration. The second section discusses several notions of architecture and model abstractions related with enterprise service architectures. The third section describes approaches to identify new and existing services, including the identification of new services based on process patterns. The fourth section describes techniques that are related to process pattern identification. They include process model comparison and querying. Since the proposed process pattern matching and discovery techniques in this work are based on graphs, the fifth

section focuses on graph matching and frequent subgraph discovery techniques, including a number of approaches beyond the scope of SOA design and development. The chapter finishes by summarising some weaknesses and limitations of the state of the art, which defines where this work builds up on.

2.2 Introduction to Enterprise SOA, EAI and BPM

Software applications are built or acquired to provide specialised functionality required to support business processes. If new activities and applications are created and integrated into existing business processes and infrastructures, new architecture and information requirements need to be satisfied [Themistocleous 2004]. Enterprise Application Integration (EAI) aims to link separate applications into an integrated system driven by business models and the goals they implement [Linthicum 2000], [Gorton 2004].

Business process management (BPM) aims to improve productivity, product quality, and operations of an enterprise. BPM encompasses methods, techniques, and tools to support the analysis, design, implementation and governance of operational business processes. Processes models have a critical role in the redesign of business processes and its integration and automation [Johannesson 2001]. However, business analysts and software developers often face difficulties managing challenges such as discovering, modelling, and understanding business processes in the context of their implementation through software applications.

Increasingly, enterprises are using Service-Oriented Architecture (SOA) as an approach to EAI. Despite the design and implementation challenges, the expected benefits have encouraged the adoption of SOA [Erradi 2006], [Linthicum 2004], [Umar 2009]. SOA has the potential to bridge the gap between business and technology and to improve the reuse of existing applications and the interoperability with new ones. Software services are the building blocks of SOA; they can be composed to provide more complex functionality and to automate business processes. However, if applications are created without a structured architectural design, integrating these into a coherent architecture closely aligned with the business processes becomes a significant challenge [Land 2007]. Often business processes do not map one-to-one to services in a process-centric service architecture, that makes refinement (and abstraction) between abstract process model levels and more concrete implemented process levels difficult to approach systematically and to automate [Koehler 2008b].

Placing BPM on top of SOA has shown good results but many challenges remain [Woodley 2005]. Short term goals can diminish the potential benefits of a strong architectural governance function [Brahe 2007], including design knowledge reuse. Abstraction and knowledge representation are principles that can address these chal-

lenges. Architecture abstractions like patterns and styles capture design knowledge and allow the reuse of successful applied designs, thus improving the quality of software [Monroe 1997]. Abstraction is a central driver in software engineering approaches; at the business level the reuse of successfully business designs is equally important. The development of integrated enterprise-wide application architectures is a continuous process. To improve the process and overall quality, the experience of analysts, architects and developers should be captured and reused. Knowledge gained from integration projects should be captured to build a repository of experience-based pattern solutions.

2.3 Architectural Abstractions in Enterprise SOA

2.3.1 Enterprise SOA Design Approaches

Two early service-oriented development methodologies, SOMA[1] (Service-Oriented Modelling and Architecture method) [Arsanjani 2004] and MSOAM (Mainstream SOA Methodology) [Erl 2004] originated from the IT industry sector. Both methodologies encompass an enterprise-wide vision and during analysis stages they cover business and applications architecture modelling domains. In MSOAM, the analysis of business-level models starts focusing mainly on processes; while in SOMA, it starts with functional areas of the enterprise. Initially, none of the two methodologies provided a detailed and integrated modelling framework and neither of them discussed support for inter-organisational processes and services.

Later, the standardization institution OASIS proposed a methodology to design business-centric service architectures [Jones 2005]. The methodology focuses on enterprise project levels and it provides a basic notation to model services. The notation is not formally defined and allows an abstract definition of business-centric services. Detailed design and implementation is beyond the scope of the proposed notation. The analysis phase from where service designs are derived included only models at business levels, leaving out of scope the study of existing applications supporting the operation of the business. Functions derived from the division of work among organisational units are the basis to define services functionalities. After services are defined, business processes are considered abstract guidelines to define service orchestrations.

Subsequently, from academia, an empirically-based methodology for SOA development was proposed [Papazoglou 2006a]. The proposed methodology is a syn-

[1]The same author participated in an earlier proposal consisting in a goal-driven approach to enterprise component identification and specification [Levi 2002]. A goal-oriented model of a business is created and developed into a business architecture and then mapped onto a component-based software architecture.

thesis of existing methods and techniques used for SOA development. It includes planning, analysis, design, construction, testing, provisioning, deployment, execution and monitoring phases. During the analysis phase, the use of reference models and documented design knowledge (for instance, standard processes) is emphasized; however, it is not clearly established how this can be operationalised. Standard modelling notations are suggested for models at different stages. However, to facilitate the overall lifecycle of services, an integrator modelling framework for all stages is needed.

At enterprise scale, the emphasis on business modelling as a necessary step in the development of services has promoted methodologies such as Business-driven Development (BDD) [Mitra 2005]. BDD is focused on the idea of developing software based on abstract business process models that can be transformed into executable processes – composed of services. The benefits promoted by BDD are agility and flexibility to deploy new software solutions. When changes at business process level take place, a direct transformation to lower levels can be deployed quickly. The latter strategy has also been promoted by several other proposals that aim to translate business process models into executable processes (e.g., [Gardner 2003], [White 2005] and [Ouyang 2007]). They often focus on solutions to directly translate process models described with process modelling notations, such as BPMN [OMG 2008b] or UML-activity diagrams [OMG 2007], into executable processes in languages such as WS-BPEL [OASIS 2007]. While the majority of proposals follow an element-by-element transformation, in [Ouyang 2007] the authors introduced the concept of workflow patterns to support pattern-based transformations. One shortcoming of these approaches is that they assume software services have a one-to-one relationship with activities in business-level process models. However, software services can involve more than one activity and in some cases, they may be more granular than an activity in a process model. This observation and other considerations for business model-driven development, including concerns about constraints imposed by existing applications are discussed in [Koehler 2006],[Koehler 2008b].

The service-oriented modelling and architecture method SOMA [Arsanjani 2004] is extended to a model-driven framework to SOA design (SOMA-ME) in [Zhang 2008]. UML profiles extend the UML 2.0 metamodel to domain-specific concepts. SOMA-ME is tool supported in IBM Rational Software Architect to provide a development environment. Traceability between models, variation-oriented design and facilities to maintain a repository of patterns and allow their instantiation are relevant characteristics of the modelling framework. Service identification support consists in an automated services categorisation function based on the ordering of business functions using a clustering algorithm. A set of values for different criteria are manually assigned to each service to provide a service score that is later used for

selection purposes.

A recent contribution proposes a UML-based framework to assist the development of service-based systems [Spanoudakis 2010]. A service discovery technique allows the identification of services (used to build the system) in a repository. A query language allowing the representation of structural, behavioural, and quality characteristics of services and a query processor are proposed as basis of a service discovery technique. The framework proposes an iterative process, where structural and behavioural design models of service-based systems and additional constraints are used to identify services that can fulfill functional and non-functional characteristics of the system in development.

Table 2.1 summarises a number of characteristics of the previously mentioned SOA methodologies. The third column in the table indicates the modelling strategy used to analyse the domains involved during services design. The strategies are top-down (from business to software levels), bottom-up (from software to business levels) and in-the-middle (using models in software and business as input to design services). The fourth column indicates the modelling languages considered in each approach. The fifth column refers to modelling activities which are tool-supported or automated to some degree. Modelling activities include transformations from business to software levels (vertical transformations), architecture/model modifications (horizontal transformations) and service identification. The latter involves the identification of new and existing services. For new services this includes the analysis of processes and existing applications that can reveal opportunities for reuse and constraints for services implementation and their composition. Finally, the last column of Table 2.1 indicates if the methodology takes into consideration the use of patterns (business and software levels) as a medium to document and manage design knowledge.

Table 2.1: SOA design methodologies.

Source	Proposal	Service design strategy	Modelling language	(Semi-)automated modelling activitities	Patterns
Industry	**Arsanjani, A. (2004)**	In the Middle	UML	none	software level
Industry	**Erl, T. (2004/08)**	In the Middle	informal	none	software level
Industry	**Mitra, T. (2005)**	Top-Down	UML	none	software level
Industry	**Jones, S. (2005)**	Top-Down	informal	none	none
Academia	**Papazoglou et al. (2006)**	All	BPMN, UML	none	business level
Academia	**Ouyang et al.(2006/07)**	Top-Down	BPMN, BPEL (+PetriNet)	model transformations (↓)	business level
Industry Academia	**Koehler et al. (2006/08)**	Top-Down	UML, BPEL	model transformations (↓) architecture modifications (→)	software level
Industry	**Zhang et al. (2008)**	In the Middle	UML	model transformations (↓) service identification	business level software level
Academia	**Spanoudakis et al. (2010)**	In the Middle	UML	service identification	none

Summarising, SOA methodologies have advanced the way services and their organisation are designed and developed, however there are still challenges such as providing integration between modelling views, guidelines and tool support to design services with adequate granularity and satisfying the constraints imposed by existing applications and mediums to reuse knowledge embedded in reference models and other types of design abstractions documentation. In particular, a special section in this chapter is focused on the use of patterns as a medium to express design knowledge that can guide the design of abstract services.

2.3.2 Architectural Abstractions

The potential benefits of reusing expert design knowledge expressed in architectural/model abstractions such as patterns and styles has not been fully exploited by existing SOA design methodologies and frameworks. The previous section mentioned some approaches that use software- and business-level patterns as a medium to reuse previously applied designs. Architectural abstractions such as patterns and styles can capture design knowledge that is expressed as a set of constraints over types and relationships among elements of concrete models/architectures. These relationships can go beyond structural relationships and the abstraction level can vary. Related concepts to software architecture level patterns and styles such as architectural frames [Rapanotti 2004], method chunks [Ralyte 2008] and business process level patterns [Gschwind 2008], [Tran 2006], [Tran 2007], [Smirnov 2009], [Thom 2007] are abstractions that can also be used to assist the design of service-based systems within specific frameworks. In particular to patterns, the software and business communities are involved with diverse definitions around the pattern concept. Some of these definitions are revisited next.

2.3.2.1 Software Level Abstractions

Several concepts capturing software level abstractions, not always clearly differentiable, have been proposed. Design patterns, architectural patterns, styles, architectural frames and method chunks are examples of them.

Design patterns in [Gamma 1995] follow the concept of patterns for architectures of buildings and towns proposed in [Alexander 1977], where patterns are solutions to problems in a particular context. Patterns can then be identified with a name and contain a description of the addressed problem, how to address it (i.e., the solution) and the consequences of applying the pattern. In [Riehle 1996], the concept of pattern is used to capture and communicate software design experience. Specifically, the authors define patterns as abstractions from a concrete form which keeps recurring in specific non-arbitrary contexts. They categorise patterns in conceptual

patterns, design patterns and programming patterns. They also describe different ways to describe patterns, including the template patterns in [Gamma 1995], and alternatives to combine patterns.

Architectural patterns in [Buschmann 1996] are used to communicate the structural organisation of a software system. The architectural pattern defines a set of predefined subsystems, their responsibilities and guidelines to organise the relationships between subsystems. In a recent volume of the pattern-oriented software architecture series books [Buschmann 2007], the authors discuss (in the context of software development) what patterns are, what they are not, and how to use them. According to them, patterns document existing best practices built on tried and tested design experience. Patterns identify and specify abstractions that are above the level of single objects, classes and components, and they provide a common vocabulary and shared understanding for design concepts. Patterns can be a means for documenting software architectures, support the construction of software with well-defined properties, and capture the experience in a form that can be independent of specific projects, implementation paradigm, and often even programming language. In the view of the authors, developers tend to see patterns as fixed blueprints or very specific configurations of classes, and they may believe that patterns help them to formulate intricate architectures by following a recipe or through mechanical application. A limited or misunderstood pattern vocabulary may cause developers to apply the wrong pattern in response to a given problem. Patterns are not just pieces of neat design, they require reoccurrence and generality. They are neither coding guidelines nor components.

For the authors in [Avgeriou 2005], architectural patterns and architectural styles are in essence the same concepts and they only would differ in that they use different description forms. They use the term architectural pattern as an umbrella term to cover the classical idea of an architectural pattern as in [Buschmann 1996] and concepts of architectural styles (described in the next paragraphs). Also, design patterns as in [Gamma 1995] are considered to be at a lower level (system level) of abstraction. Thus, in [Avgeriou 2005], architectural patterns and styles would refer to recurring solutions that solve problems at the architectural design level, help to document the design decisions taken by architects, provide a common vocabulary facilitating communication and the means to reason about the quality attributes of a software system. However, styles would focus less on the problem and rationale behind selecting a specific solution. According to the authors, the lack of consensus in the community with respect to the ”philosophy” and granularity of architectural patterns, as well as the lack of a coherent pattern language make difficult to apply the appropriate architectural patterns in practice. In an attempt to find a common understanding, the authors in [Avgeriou 2005] propose a pattern language that acts

as a superset of the existing architectural pattern collections and categorisations.

For the authors in [Gregory 1995], software architecture styles are seen as a collection of conventions used to interpret a class of architectural descriptions. Thus, in order to understand the meaning of a specific architectural design two elements have to be present: a description of the design (usually in the form of an architectural diagram) and a reference to the style used to describe the design. A style indicates the kinds of components in the design diagram, their relationships and other semantic details, such as constraints on the topology. The authors in [Gregory 1995] describe styles by means of mappings from a syntactic domain of the architectural description to a semantic domain of the architectural meaning. In [Taylor 1996] a particular style (C2) is proposed, which allows to model distributed and concurrent applications. In [Pahl 2007], an ontology-based approach for modelling architecture styles is presented. The authors introduce operators for style modification and combination between styles. Relationships between quality requirements and style modelling are investigated. The application of the ontological framework in service ontologies is illustrated.

Considering the scope of styles in [Gregory 1995], [Taylor 1996], there is not a clear boundary between the definition of a style and an architecture description languages (ADL) [Medvidovic 2000]. In a similar manner, architectural patterns, design patterns and styles are not clearly differentiated. For instance, in [Taylor 1996] model-view-controller (MVC) is considered a style, while in [Curry 2008] it is considered a design pattern, in [Shaw 1996b] an architectural pattern (implicit invocation architectural pattern), and in the original publication [Glenn 1988] is considered a programming methodology. In [Giesecke 2007], the authors generalise architectural styles, patterns and similar concepts by introducing the notion of architectural constraints. An architectural constraint is considered a medium to reuse architectural design knowledge and to improve software quality.

Architectural frames, which are combinations of architectural styles and problem frames are introduced in [Rapanotti 2004]. Problem frames classify software development problems and, therefore, are used only in the problem space. The authors observe that the solution space influences the problem analysis within a domain through the software engineer's domain knowledge, including choices of domain-specific architectures, architectural styles, development patterns and reuse of past development experience. Architectural frames use architectural styles, located in the solution space, to guide the analysis of the problem space.

The structure of a knowledge-based system for situation-specific solutions, called method chunks, is proposed in [Ralyte 2008]. Method chunks are based on the ideas of modularisation and formalisation of method knowledge in the form of reusable entities from situational method engineering. Method chunks can be combined to

compose a situation-specific method and can be stored in a repository. An interoperability classification framework is used to classify and tag method chunks, and to assess the project situation in which they are to be used.

This work is centred on the idea of patterns, and the word *pattern* would be used to refer to a set of constraints over elements and their relations in concrete models, including architectures, that allows to capture a known solution to a recurring problem. This view is similar to the notion of architectural constraints in [Giesecke 2007] but focused on the problem-solution pair associated to architectural patterns and styles in [Avgeriou 2005].

2.3.2.2 Business Process Level Abstractions

Different variants of business process level patterns are introduced in [Tran 2006], [Tran 2007], [Barros 2007], [Gschwind 2008], [Smirnov 2009] and [Thom 2007]. Also, basic workflow patterns for control flow are widely known and available in numerous process modelling tools [Aalst 2003]. This section provides an overview of these variants.

Workflow patterns constrain the flow of information in processes that can be executed as workflows [Aalst 2003]. Examples of them are exclusive choice, parallel split, simple merge and synchronization patterns. Yet workflow patterns are widely known and available in numerous process modelling tools, few of them provide support for end user to correctly apply these basic patterns. The authors in [Gschwind 2008] describe the extension of a business process modelling tool with workflow patterns. The tool supports the use of pattern compounds and provides active recommendations for selecting patterns that are applicable in some user-determined context, provides feedback to the end user if applying a pattern can lead to a modelling error and it allows to trace the sequence of applied patterns during the model editing process. More complex patterns that involve the semantics of a business domain or a formal conceptualisation of process patterns is not addressed.

An approach to formalise the concept of process pattern based on a process metamodel is introduced in [Tran 2006] and [Tran 2007]. The authors provide a general definition to cover various kinds of process-related patterns in different domains and a method to construct models based on pattern combinations. They include generic process structures and patterns for determined domains such as the *stage process pattern*, *process interaction*, *time-to-customer* and *process layer control* in [Ambler 1998] and [Eriksson 1998], as well as process patterns applicable to specific software systems, often captured from internal processes in organisations (e.g., they mention Fagan's process for software inspection [Fagan 1976]). In [Barros 2007], business process patterns are considered the medium to encapsulate high-level business knowledge

and logic of a given application domain. Business process patterns would capture best practices and they could be reused to improve processes quality. Examples are the process patterns documented in the Supply Chain Operations Reference-model (SCOR), the Enhanced Telecom Operations Map (eTOM) and the Information Technology Infrastructure Library (ITIL). In [Barros 2007], a number of real process redesign projects (hundreds) were studied, and business patterns extracted. In [Smirnov 2009], the authors address the problem of providing suggestions during process modelling with the objective of assuring model quality and homogeneity. They introduce action patterns (chunks of actions often appearing together in business processes) and a technique to identify them. Actions patterns are domain specific and they are identified from existing process model repositories using association rule mining techniques (the Apriori algorithm [Agrawal 1994]) and based on the similarity of activity labels. Action patterns can then be used to suggest additional actions for a process model. The authors use the SAP Reference Model to illustrate the type of pattern that action patterns are and to evaluate their approach. In [Thom 2007], the authors describe three main categories of workflow patterns capturing recurrent business functions frequently found in business processes: workflow patterns based on *organizational structural* aspects, *specific application domains* and *recurrent functions*. The latter category is independent of the application domain. In one of their recent works [Thom 2009], the authors introduce *workflow activity patterns* (WAP) – or *activity patterns* for short – to capture descriptions of recurrent business functions found in business processes such as the *task execution request*, *notification* and *approval* patterns. Two main categories are considered: activity patterns based on *organizational structural* aspects; and *recurrent functions*. Activity patterns are described in a similar way to software design patterns. A structured document is used for this purpose and it contains fields for a pattern name, a textual description of the pattern and addressed problem, an illustrative example of their use, design choices that determine different pattern variants, references to related patterns and remarks regarding pattern implementation. Two hundred process models were analysed in [Thom 2009] in order to evidence the practical relevance of process patterns.

The concepts of activity patterns based on recurrent functions in [Thom 2009] and actions patterns in [Smirnov 2009] are close to the concept of *process pattern* in this work. In the same way software level patterns define a set of constraints over elements and relations among architecture elements, *process patterns* define constraints over process elements and their relationships in concrete models. They abstract recurrent (process) steps across processes and organisations. However, process patterns are used for a different purpose than in [Thom 2009], [Smirnov 2009]. In a similar intention to [Papazoglou 2006a], [Papazoglou 2006b], where the authors suggest the use of reference models and associated business-level patterns to guide the defi-

nition of normalised business functions for services, this work uses process patterns identified in process level descriptions to guide the design of reusable process-centric services, which would be used in integration systems. The proposed approach aims to advance techniques for pattern-based service definition by providing an automatic mechanism for process pattern identification.

2.3.3 Pattern-based Techniques

Different techniques can be used to benefit from patterns and to assist the development of systems. Techniques can support the selection of appropriate patterns and their adequate instantiation in developing systems. They can also assist, in a controlled way, the transformation of existing systems. Moreover, if patterns have been previously applied but the architecture of the system has no documentation about them, recovery techniques can assist the extraction of higher level architecture models. Similar to software levels, process models can be extracted from process execution logs, the application of process patterns can be assisted during process modelling, and process patterns can be identified in concrete models to assist the development of pattern-based systems. Towards the end of the chapter, pattern matching and discovery techniques, which are core to this work, are described in the context of processes and graph-based models.

Pattern Selection. A systematic method to select patterns using pattern language grammars and design space analysis is introduced in [Zdun 2007a]. The aim is to facilitate the decision to select the appropriate patterns and pattern languages during the construction of systems based on patterns. Pattern languages prescribe the way patterns can be combined, defining interdependencies between patterns and often documenting extensive examples or case studies explaining the combined use of patterns. The authors explain that difficulties arise when patterns and pattern languages are written by multiple entities, making necessary to identify interdependencies and overlaps between patterns and pattern languages before their utilisation. The pattern selection in [Zdun 2007a] is based on the desired system's quality attributes. Pattern relations are formalised in a pattern language grammar and the grammar is annotated with effects on quality goals. Design decisions are analysed using the design spaces covered by a set of related software patterns, possibly originating from different sources.

Design decisions for service-based systems development have their own particularities. In [Zimmermann 2007], a method to design solutions for service-based transactional workflows is presented. The method identifies recurring architectural decisions in analysis-level process models, models alternatives for these decisions

as reusable, platform-independent patterns and primitives, and maps the patterns and primitives into technology- and platform-specific settings in WS-BPEL and SCA. Complementary to [Zimmermann 2007], in [Papazoglou 2006b], a design approach for business transactions based on standard business functions is proposed. Both methods guide the design of service- and process-centric systems using patterns at software and business levels, respectively. Guidance however requires the definition of mappings from problem to conceptual primitives/patterns and from them to known technical solutions in platform specific settings. Clear identification of the design problem requiring a decision and posterior analysis of previous applied patterns needs to be managed outside the proposed methods.

A few contributions have addressed the problem of pattern applicability in an automated way. In [Kim 2007], an approach to specify the problem domain in design patterns is proposed. The domain problem of patterns follows the ideas for pattern descriptions in [Gamma 1993], but adding a precise notation. Their conceptualisation and precise notation of the problem domain of patterns is suggested for the utilisation of tool support for automatic evaluation of pattern applicability. They demonstrate how the problem domain specification of the visitor pattern can be used to evaluate pattern applicability in a more automated way.

An automated identification of process patterns in high-level process descriptions (involved in the development of service systems for process and application integration) is proposed in this work as an alternative to automate the identification of the problem domain of services in development, which are subsequently refined to service architecture implementation levels. This complements methods such as [Zimmermann 2007] and [Papazoglou 2006b] with automated assistance during services identification.

Pattern Variation and Combination. Non-controlled changes in an architecture might interfere with previously applied design patterns. Several proposals in the literature have taken advantage of the graph representation of software architectures and they have used a graph basis to address issues on pattern evolution. In [Zhao 2007], a graph-transformation approach to pattern level design validation and evolution is presented. Based on types of design pattern evolution, they specify graph-transformation rules to manipulate the pattern elements while maintaining the underlying pattern properties of the design. The authors do not refer to pattern property preservation after pattern combination.

Patterns are not often used in an isolated way, they can be part of organised collections named pattern languages. Pattern languages allow regulated combinations that extend the reach of individual patterns [Buschmann 2007]. In [Hentrich 2006] a pattern language for process-oriented integration of software services is presented.

However, it is only focused on structural aspects of executable processes, such as the synchronisation between macro- and micro-process flows. The development of architectures with a pattern-based approach requires techniques providing mediums to combine, refine, extend, and other more complex operations to work with patterns. In [Gomes 2003] a set of pattern operators is introduced to design architectures for applications in grid environments. In [Gomes 2008] these ideas are extended to grid-based workflows.

Pattern Recovery. Architecture modifications are central after a software system is implemented. Discovering the instances of previously applied patterns and providing techniques to modify the architecture in a controlled way are two important activities in software maintenance.

Since the nineties, pattern discovery techniques have been proposed to recover patterns from source code. Discovering instances of architectural and design patterns from the source code of software systems is used to assist the analysis of systems and their future modifications. Source code is typically large in size, making more difficult its comprehension. Trying to manually identify patterns from the source code consumes significant time and it is susceptible to errors. Most pattern recovery approaches first extract a class model from the source code and subsequently analyse that model in the search of patterns. The identification of architectural/desing patterns in class models mostly relies on matching structural characteristics – from patterns present in models. Less considered are the behavioural and semantic aspects.

A technique to detect design patterns using a similarity scoring approach is introduced in [Tsantalis 2006]. The technique is flexible to allow the identification of modified versions of patterns. The technique is adapted from a previous graph-based algorithm for ranking pages on Internet [Jon 1999] and exploits the fact that patterns reside in one or more inheritance hierarchies, reducing the size of the graphs analysed by the algorithm. An evaluation in three systems indicates that the technique produces only a few false negatives and no false positives in their results. In [Sartipi 2001], a technique for recovering the high-level design of legacy software systems based on a pattern matching technique is proposed. The technique is a based on a graph matching algorithm that takes as inputs a graph-based representation of architectural patterns defined by end users and a graph representation of legacy systems. A multi-phase branch and bound search algorithm with a forward checking mechanism controls the matching process of the system and pattern graphs. For process pattern identification, a representation of the semantics involved with process elements and behavioural aspects needs to be added to the structural aspects addressed in [Sartipi 2001].

A review of 26 design pattern discovery techniques can be found in [Dong 2007]. The authors also propose a design pattern identification technique that considers structural, behavioural and semantic aspects. The review indicates that the choice of source code representation in specific types of models directly affects the choice of the algorithms for discovery. Some techniques only allow exact matches whereas others may allow approximate matches. Most techniques provide as their result only the number of discovered patterns, whereas some of them indicate the positions of discovered patterns graphically. But more restrictive, in general, the discovery techniques only support the discovery of a certain number of patterns. The proposed technique in [Dong 2007] uses weighted matrices to represent the system architecture and patterns. Patterns are discovered by comparing matrices associated to the architecture and patterns. Instances of different patterns in a single architecture model which have a similar structure and behavioural representation in matrices can be only be distinguished by reviewing if the pattern elements were named with some pattern-related information. Naming conventions of classes have to contain references to design patterns if instances of different patterns need to be distinguished.

2.3.4 Traceability in SOA Modelling

Traceability among elements of an architecture is an important property for architecture modification and evolution. Traceability between software components and relevant business elements of an enterprise has been exploited as a fundamental instrument to manage complex enterprise systems [Bernus 2003]. Service-based systems for business processes and application integration are at the core of an intermediate layer between models representing business operations and software architectures. Traceability between business elements, services and software components is important to manage the consistency between views involved in a process and application integration problem.

The authors in [Steen 2005] discuss the relevance and impact of service orientation to enterprise architectures. Traceability between services is utilised as instrument to align different views and to analyse the impact of changes in a single view. Even though modelling support for traceability is provided, the architectural description of services is introduced in few details.

A multi-viewpoint approach for service-oriented design is introduced in [Dijkman 2004]. The authors focus on interrelations of viewpoints at service and service composition level to allow consistency between different parties designing an inter-organisational service-based architecture. The views involve interface behaviour, provider behaviour, choreography, and orchestration. They formalise models from a control flow perspective using Petri nets. A traceability model interrelating

views is used to allow static verification of the consistency of composite services.

Practical experience regarding the adoption of a service-centric architecture in the context of a very large and complex enterprise transformation is described in [Kavianpour 2009]. The publication reveals the method used in Unisys to manage the complexity of an enterprise-wide architecture transformation process, in this case to a service-centric architecture. Models capturing different views are interrelated in a traceability model. The traceability model is manually maintained. The objective of the traceability model is to assist a change impact analysis by observing the effects of changes from one model over models in other views. Changes in models are intentionally made to emulate possible changes in reality. The effects on the overall architecture are analysed and used to assess possible architecture configurations. The analysis is performed by end users and facilitated with the information of the traceability model.

An initial version with tool support involving functionality to manage traceability among models in the SOMA-ME modelling framework for service systems development is proposed in [Zhang 2008]. Predefined relations between model elements define specific structures that are maintained and monitored. If changes in elements occur, preservation of the predefined structures is checked and if alterations are found, alerts to the end users are displayed using a colouring scheme on model elements. This approach allows interaction with end users to maintain consistency between models after changes. In [Zhang 2009], the initial work in [Zhang 2008] is extended to facilitate the use of SOA solution patterns in concrete models associated to a service-based system in development. Solution patterns can be associated to industry standards or best practices. During modelling, end users with the help of the provided tool support can select solution patterns to be applied into their models. The history of applied patterns on particular models is recorded and it constitutes a traceability model between patterns and concrete models.

2.4 Service Identification

A critical step to design service architectures is service identification. Service identification can involve the discovery of existing services or modelling new ones. High-level guidelines to design new services as the ones mentioned in the previous section (e.g., [Erl 2004]) are very useful, however they require advances regarding formality and techniques that can be realised in tool support. The following sub-sections describe a number of approaches providing techniques to (semi-)automate steps in the service identification activity.

2.4.1 Identifying New Services

During early stages of *services design*, service granularity is decided according to principles such as loose coupling, reusability, abstraction, autonomy. Most SOA design methodologies consider services design a highly human-dependant task. Some contributions, such as [Albani 2006], [Aizenbud-Reshef 2007] and [Zhang 2005] have proposed techniques to (semi-)automate the identification of software components and services.

A semi-automated method to identify *business components* from an enterprise ontology is proposed in [Albani 2006]. Business components are used to automate inter-organizational business processes, combining different software artifacts. They can be seen as precursors of business-centric services. The business domain model from where business components are derived from satisfies the requirements of an enterprise ontology. A matrix-like structure derived from the enterprise ontology is used to organise business process steps from process models and manipulated objects in each of the process steps. Each matrix entry has a value related to an operation type. Operation types capture the different ways an object can be manipulated (processed) in a process step. After the matrix structure is (manually) populated, it is re-organised using an optimisation algorithm that generates clusters. These clusters are used to guide the definition of software components. Optimisation criteria aiming at minimal communication between components (loose coupling) and maximum component compactness (high cohesion) are used to identify the clusters. The service identification method in [Albani 2006] can be seen as a top-down (from business to software) method.

A bottom-up, semi-automated method to identify service candidates in legacy source code is proposed in [Aizenbud-Reshef 2007] (associated with patent application [Aizenbud-Reshef 2009]). The idea is to abstract information from legacy source code, store that information in a repository and to match a service description-based query with source code elements described in the repository. A ranking engine can combine matches in order to obtain an aggregated answer to the service description-based query. Information retrieval techniques and static analysis are used to retrieve ranked candidate locations according with semantic context and similarity between functional description associated to services in the query and functionality of source code in the repository.

The approach in [Zhang 2005] focuses on identifying services based on a comparative analysis between information extracted from the problem domain at the business level and architectural information derived from legacy systems. Architectural information is recovered from legacy systems to subsequently apply a (hierarchical) clustering technique to group legacy functionality from different systems.

Existing (grouped) functionality in legacy systems is matched with functionality of ideal services derived from the business domain. Ideal services capture functional requirements for concrete service implementations. They are derived manually from the analysis of the business domain. The analysis aims to identify opportunities for reuse across the domain.

2.4.2 Identifying Existing Services

The previous contributions to service identification focus on the identification of *new* services. A related problem is the identification of existing services satisfying specific requirements. This is of particular interest in service composition. There are a number of approaches to identify existing services, from proposals based on matching service signatures and effects to matchmaking techniques involving complete process-centric service descriptions, from pure syntactical based approaches to semantic ones [Kuster 2008], and from high-level process service descriptions to low (implementation) level services (e.g., web services [Dustdar 2005]).

Approaches involving complete process-centric descriptions are motivated by the idea of service requestors requiring detailed information of how the service processes the input information (messages) and changes the state of system in which it participates. Also, they are motivated by the fact that exact matches may be unlikely to exist so that results similar to the expected ones are also relevant. Semantic matching approaches aim to advance the lack of relevant results that more syntactical proposals provide. Semantic descriptions involve information used to capture the meaning of the described elements (services) and allow reasoning about them. However adding semantic annotations to services and the existence of a common ontology is difficult to realise in practice, for example, due to the distributed and heterogenous nature of entities describing the offered services.

Rather than providing a comprehensive description of existing service identification techniques, a number of contributions focusing on matching techniques involving complete process-centric descriptions are explained. Process descriptions include descriptions of *existing* services but also business processes, which are the problem domain during the identification of *new* services.

Process centric descriptions and the matchmaking process in [Wombacher 2004] are based on a formal semantics of deterministic finite state automata extended by logical expressions associated to states. A matching function calculates the intersection between annotated automata representing the required and offered services and checking the emptiness of the resulting automaton. In [Corrales 2006] (and a subsequent contribution in [Corrales 2008]) services from a repository are identified using a graph-based technique to match behaviour. Services described in WS-BPEL

in a repository and queries to it are described as graphs. Existing graph edit distance algorithms are adopted to perform an approximate matching step. The result of this step is a ranked list of similar services from the repository. The computational complexity of the proposed matching technique ranges between $O(m^2n^2)$ and $O(m^n n)$, with m and n the number of vertices from compared graphs representing a service in the repository and a service query. In [Eshuis 2007] various types of structural matches for BPEL processes supporting dynamic binding of services are defined. BPEL processes are modelled as process trees, where each tree node is an interaction. Activities that are not interactions are abstracted into internal steps and cannot be matched. Duplicate interaction activities are not allowed in the tree. *Plugin* matching is presented as an approach based on a process simulation notion. However, as the authors indicate, the proposal requires further semantic analysis to decide if a process can replace another after matching. Similar to [Corrales 2006], the efficiency of the matching technique in [Wombacher 2004] can result in scalability problems. Computing the intersection between automata in [Wombacher 2004] can be expensive and present difficulties when cycles are involved. Another matching approach, also based on service automata, is presented in [Massuthe 2007]. Services are modelled as open workflow nets and their behaviours described as a service automata. Based on arbitrary finite-state service automata, the concept of an operating guideline is introduced, which gives complete information about how to properly interact with an open workflow net. A service requestor interested in a service composition can focus on matching operating guidelines associated to offered services. These are limited to deadlock-free interactions between services in the composition and bounded message buffers restricting the communication. The matching process is performed in a single depth-first search through a deterministic automaton representing the service requestor.

A recent service discovery technique was proposed in the context of a framework to assist the development of service-based systems [Spanoudakis 2010]. A query processor is used to match service queries against services in a registry. The matching process involves matching of signatures, behavioural models and soft constraints. Signature matching is based on a function combining linguistic distances between the names of the operation and query messages, the names of their parameters, and the data types of these parameters. Behavioural matching is based on matching state machines representing the behaviour associated to interfaces of a query service (SMQ) and a service in the repository (SMS). The matching relies on the minimum sum of distances between possible pairs of transitions, where pairs of transitions are composed of a transition in SMQ (linguistically) similar to a transition in SMS. State machines are generated automatically from an interaction diagram representing the behavioural model of the service query and a WS-BPEL specification of the offered

service. State machine structures or states are not compared. Data type graphs which have more than 10 edges are considered graphs with medium to high complexity. The evaluation of the overall querying approach indicates good results for recall and precision. However, as indicated by the authors, although the processing time grows linearly with the size of the service registry, service retrieval is slow. They refer to an appropriate indexing schema or alternative DBMS for future implementations of the service registry in order to improve performance.

2.5 Process Models Comparison and Querying

Several approaches to service identification involve comparing or querying process-centric service descriptions. A number of contributions have specifically targeted the problem of business process models comparison and querying.

An algorithm for calculating process similarity to cluster open-source process designs is presented in [Huang 2004]. Processes are composed of services related through control flow elements (links). A weighted graph is used to represent processes. A graph similarity measure is proposed. It is based on the weighted sum of similarities between sets of services and sets of service links in a process. Service similarity involves several similarity measures that include comparison of semantic information of services, operations that can be invoked, messages and data exchanged. These measures are adopted from [De Antonellis 2003], [Intan 2002].

A method to measure structural distance between process definitions associated to web services is presented in [Bae 2006]. The method relies on a distance measure of normalised matrices representing graph-based process models. Improvements on the data structure for matrices could provide more flexibility to represent processes and improve performance. In [Beeri 2008], the authors propose a query language for WS-BPEL process descriptions based on Context Free Graph Grammars (CFGG), which in general are not closed under intersection. Process queries are graphical queries annotated in the same way as process descriptions. Activities can be zoomed-in by means of graph refinement. Cycles in process graphs and graph refinements containing recursion are handled by representing compacted graph structures. Replacement in the utilised CFGG consider isomorphic relations between graphs. Structural relations between processes and queries involving surjective graph morphisms are not allowed. Many fork (split) and join constructs in service descriptions (i.e., a high in/out degree in vertices from graphs representing process descriptions) could lead to an exponential number of paths in the result of a query. Matching of graph vertices requires syntactical equivalence of vertex descriptors. A simple descriptor is a vertex label. Extensions to consider label predicates and regular path expressions are discussed.

The authors in [Aalst 2006] propose a way to compare two process models based on their observed behaviour. Observed behaviour relies on the information extracted from logs of process executions. Mining techniques are applied over sequences of process steps [Aalst 2007]. While the focus of this work is rather on matching graphs representing process models, the results of mining techniques (i.e., graph-based models representing sets of process executions) can be considered as input to the addressed problem and, therefore, they are complementary techniques to the addressed matching and discovery techniques in this work.

An approach using digital logic is used to evaluate the distance and similarity between two process models based on high-level change operations such as add, delete or move activities [Li 2008]. Because of the complexity of the technique, the aim is to minimise the number of high-level change operations needed to transform one process model to the other. High-level change operations are adopted from [Reichert 1998], [Weber 2007]. In [Li 2009] the authors focus on adaptive process-centric systems. This type of system enables structural process changes during runtime. Changes produces process variants that are expensive to configure and maintain. A heuristic search algorithm which fosters learning from past process changes by mining process variants is proposed[2]. The algorithm discovers a reference model based on which the need for future process configuration and adaptation can be reduced. The authors indicate that the algorithm takes a relatively long time when encountering large process models and, therefore, performance improvements are needed.

An approach to calculate behavioural similarity between process models based on causal footprints is proposed in [Dongen 2008]. Causal footprints are an abstract representation of the behaviour captured by a process model. Similarity between causal footprints is based on vector space models used in information retrieval. Semantic similarity between labels in process elements and contextual similarity are taken into account. Similar to other contributions in process model comparison, the authors indicate that practical performance issues with large set of models need to be addressed. Later, in [Dijkman 2009b], [Dijkman 2009a], the authors provide additional details and complementary techniques to measure the similarity between business process models. The existence of business process models repositories is assumed. The focus is on retrieving process models in a repository that most closely resemble a given process model or fragment thereof. In [Dijkman 2009b], three similarity metrics are proposed. Label matching similarity that compares the labels attached to process model elements, structural similarity that compares element labels as well as the topology of process models, and behavioural similarity that compares

[2]Note that process variant mining is different from process mining [Aalst 2007] with regard to objectives and inputs.

element labels as well as causal relations captured in the process model. Findings indicate that structural similarity slightly outperforms the other two metrics. Because the proposal focused on developing similarity metrics rather than efficient algorithms, the authors indicate that future work would address efficient algorithms for searching similar processes in processes collections. Also, they indicate that rather than matching process elements one-to-one, they would investigate many-to-many matchings. This kind of matching is relevant for models that have been created by different entities. In [Dijkman 2009a] four graph matching algorithms for process similarity search are proposed (greedy, exhaustive, heuristic and A-star). The greedy algorithm has a processing time $O(N^3)$ with N the number of vertices of the largest graph in the repository and it may lead to suboptimal mappings. The exhaustive algorithm with pruning has exponential complexity in the worst case; however, the pruning stage can manipulate its complexity. The process heuristic algorithm is a variation of the exhaustive algorithm with an improved pruning phase. The A-star algorithm is based on [Bunke 1997], which uses an error-correcting strategy based on a graph edit cost function. The evaluation provided in [Dijkman 2009a] indicates that the A-star algorithm outperforms the others in terms of accuracy, however the greedy algorithm is considerably faster than all others. All algorithms only find one-to-one matches between vertices in the target and query graphs. The authors plan to investigate adaptations of the algorithms for one-to-many or many-to-many matches between vertices.

A number of proposals, including some of the previously described [Wombacher 2004], [Corrales 2006], [Spanoudakis 2010], [Beeri 2008], [Dongen 2008], [Li 2009] share the challenge of improving the efficiency of the matching processing step. Matching of graphs or state machines/automata can be computationally expensive. Service identification proposals have to look at this step if they consider a matching step as part of the solution. In this work graph-based matching and discovery techniques are proposed. In subsequent chapters, the techniques shall be described and evaluated. Without going into detail, it can be indicated that the proposed technique is efficient for graph-based structural matching steps and depends on the complexity of the adopted approach to semantic matching of vertices descriptors.

As in [Dijkman 2009b], semantic matching of process elements has been investigated in [Ehrig 2007] and [Gunay 2007]. They are motivated by the possibility that business processes can be modelled in different ways by different modelers even when utilising the same modelling language. In [Ehrig 2007], a method for solving ambiguity issues in process models caused by the use of synonyms, homonyms or different abstraction levels for process element names is addressed. They adopt ontology-based descriptions of process models – in this case, OWL DL-based de-

scriptions of Petri nets – and they address the particular questions of how similar terms for process element names can be automatically discovered and how semantic business process composition can be facilitated. Similarity calculations consider ontological reasoning with regard to business rules. Their findings indicate that syntactic and linguistic similarity are insufficient since the process instance context is not considered and homonyms cannot be discovered. However, by considering structural aspects of process instances, sufficient similarity degrees between element names and processes can be computed. Integration of the approach with other techniques for comparing processes based on control flow semantics is part of their future work. In [Gunay 2007], structural and semantic similarity metrics for web service matchmaking are proposed. Service matchmaking is based on the internal process of services. The internal processes are modelled as finite state machines. Several heuristics to find structural similarities between finite state machines are proposed. Also, a process ontology is used to capture the semantic relations between processes. Semantic information is used to determine semantic similarities between processes and to compute match scores of services. Complexity problems in the approach relate to their calculation of all possible flow sequences for services and requests and the comparison of each sequence of the request against all the sequences of the service. Also, there are practical problems with having a common ontology describing the process elements and relations associated to services.

An indexing mechanism to create hierarchical ontologies for process models and a query language to perform matchings on the ontologies are proposed in [Klein 2004]. The proposed language is named Process Query Language (PQL) and it considers process models as entity-relationship diagrams in which entities such as tasks are characterised by attributes and connected by relationships such as has-subtask. Entity-relationship patterns define queries in PQL. The retrieval approach in PQL is similar to work on graph grammars in [Ehrig 1999b] and complexity associated to the PQL's implementation is the equivalent of a datalog-type language with polynomial computational complexity. Future work for the approach involves improving recall when semantic differences between processes exist. Ideas for improvement include synonym-matching techniques and semantics-preserving query mutation, which modifies a service query to produce a range of semantically similar variants. The latter is applied in [Awad 2008b], where process queries in a process query language (BPMN-Q) are expanded by substituting activities of a query graph with similar activities. A visual process query language based on Business Process Modelling Notation (BPMN) is proposed in [Awad 2008b] to retrieve process models from a repository ordered by relevance to the query. Calculation of structural similarity between process models proposed in previous work [Awad 2008a] is expanded in [Awad 2008b] to allow semantic similarity calculation. Enhanced

Topic-based Vector Space Model (e-TVSM) is used to capture semantic similarities of natural language in plain text documents and to reflect them in document similarity values. Semantics is encoded in an e-TVSM ontology. Similarity thresholds control the expansion of queries to allow improved recall, however low thresholds tend to generate an exhaustive search in all processes from the repository, possibly leading to performance problems when queries have a large set of activity nodes (vertices). The performance problem originates from the generation of expanded queries, which is non-polynomial.

Other approaches addressing process similarity focus on clustering similar processes from a repository rather than compute specific comparisons triggered by process queries. In [Jung 2006], [Jung 2008] the authors focus on the analysis of accumulated workflow process models and their classification into characteristic groups. Domain classification and pattern analysis are the two main aspects of a proposed framework for workflow clustering. Process models are represented as weighted Complete Dependency Graphs (w-CDG). w-CDG are derived from directed graphs representing process models which have added dependencies from indirect relations originated from AND or XOR splits in process models. Similarities among graph vectors associated to w-CDGs are estimated with regard to the relative frequency of each activity and transition in the compared processes. Models are clustered based on the similarities by a hierarchical clustering algorithm. The number of process clusters is specified by end users. Semantics in process element descriptions is not considered.

2.6 Graph-based Pattern Matching and Discovery

Several practitioners and some in academia argue that changes of business-level services are less frequent as compared with more frequent changes in business processes. For instance, the authors in [Woodley 2005] use a typical example to illustrate this idea. They refer to the constant need to perform customer verifications, send shipping orders, and prepare invoices in industry processes. However, they point out that products, customers and the processes that integrate them change quite often. At a higher level of abstraction, a service performing a customer verification is fundamentally the same across process, only details change. In this work, the same observation is exploited to benefit from frequent and stable abstractions across processes. Business services are considered stable entities in relation to frequently changing processes. On the other hand, the fundamental idea in software engineering of patterns abstracting several specific software designs can be applied here at process level. Even though patterns can evolve, intuitively, their changes are less frequent than changes in models where they have been applied. Process level patterns

could guide the definition of process-centric services, where process patterns represent abstractions to sections in several processes across organisations and changing over time.

A number of pattern matching and discovery techniques are defined to work with graphs. For graph-based pattern matching and discovery there is a significant body of knowledge related to graph matching and frequent subgraph discovery. In a very simplified explanation, graph matching takes as input a target graph and a query graph and indicates as its output if the query graph has a structural preserving relation with the target graph and – possibly – it would indicate for what elements the relation holds. There are numerous variations considering what type of structural relation holds and what type of graphs are considered. On the other hand, frequent subgraph discovery has two major variants, one focusing on discovering subgraphs that occur frequently across a set of input graphs and the other focusing on the discovery of subgraphs occurring multiple times in a single large input graph. The following sections refer to contributions with regard to graph-based pattern matching and discovery which could be adapted to assist service identification.

2.6.1 Graph Matching

Graph matching has several years of investigation and solutions to this problem vary widely according to the specific graphs and type of matching required [Conte 2004], [Bunke 2005], [Gallagher 2006b], [Gallagher 2006a].

A survey and classification of graph matching algorithms is provided in [Conte 2004]. Algorithms are classified by class and application domain in the context of pattern recognition and machine vision. Matching is mainly classified into exact and inexact pattern matching, with most of the applications in image processing, and with more than 160 publications analysed. Almost 40% are algorithms for tree search. One interesting algorithm not based on tree search is the Nauty algorithm [McKay 1981]. The algorithm (used in the context of a graph-transaction setting) is based on its automorphism group, from which a canonical labelling is derived and used to introduce a vertex ordering uniquely defined for each equivalence class of isomorphic graphs. In this way, whether an isomorphism between two graphs exists can be checked by verifying if the adjacency matrices of their canonical forms are equal. This verification is done in $O(N^2)$ time, with N the size of the graph. However, for some classes of graphs the canonical labelling may require an exponential time to construct. Others – approximately 20% of the surveyed algorithms – convert the discrete optimisation problem defined by a graph matching problem into a continuous nonlinear optimisation problem. The algorithms use solutions available in the converted domain to find an approximate solution. Even though most of the

algorithms include techniques to avoid trivial local optima, they do not ensure an optimal solution. Also, the solution has to be converted back to the discrete problem, which can introduce other levels of approximation.

Several challenges related to graph matching for pattern recognition are analysed in [Bunke 2005]. The authors indicate that structural representation based on graphs have several advantages over feature vectors for pattern recognition. Feature vectors are limited because they can represent only unary properties and usually assume that the same number of attribute values is being measured on each object. String, trees and graph structures are suitable to overcome these kinds of limitations. However, they also suffer from disadvantages, including their high computational complexity and the lack of suitable mathematical tools. The authors indicate that a number of concepts emerged recently from statistical pattern recognition that have no equivalent counterpart in the domain of structural pattern recognition, including multiple classifier systems and kernel methods, that may be means to mitigate the disadvantages of graph-based techniques. In the view of the authors, some of the advances include the availability of procedures for the automatic learning of edit cost functions from sample sets of graphs. However, efficient kernel methods which are originally developed for feature representations still require advances to be used on structural data, multiple classifiers are in early stages of development and also efficient graph retrieval in large repositories is still a hard problem.

The author in [Gallagher 2006a] surveys a number of approaches to match patterns in graph-structured data. The basic matching problem is seen as a subgraph isomorphism problem. The survey focuses on techniques that are applicable to general graphs that may have semantic characteristics. The survey also discusses techniques for graph mining as an extension of the graph matching problem. In terms of performance, subgraph isomorphism algorithms are computationally expensive and therefore techniques try to reduce calculations to a minimum. An effective way to reduce processing time is candidate selection, where metadata construction and application, such as indexing and data summarization, compression, and modelling are central to an effective candidate selection. The survey indicates that existing tree-search techniques (where the graphs are trees) such as [Ullmann 1976] can be extended to match and prune based on semantics as well as structure. Existing evaluations of the surveyed algorithms are not representative of the size and characteristics of real-word graphs that include semantics. Many of the existing matching algorithms focus on a graph-transaction setting. In this setting, one query graph is matched against graphs from a repository and often the size of the query and graph in the repository are similar. Individual graphs tend to be very small and, therefore, many techniques are not directly applicable to large graphs. Exceptions are those where it is possible to divide a large graph into a set of smaller graphs. Thus,

techniques developed for the graph-transaction setting could be applied to that set. Type and attribute information in graphs could potentially improve candidate selection and indexing strategies focused on graph structure. More sophisticated graph statistics are required to capture a combination of attributes, type, and structure. Graph similarity measures used by existing techniques do not incorporate all of the attribute, type, and structural information. Combination of these different kinds of similarity is critical for inexact matching in semantic graphs.

Error-correcting is one common strategy for (inexact) graph matching. It uses a set of graph edit operations to calculate the distance between two graphs. The objective is to find the shortest sequence or sequence having the least cost of edit operations that allows to transform one graph into the other. A subgraph of both graphs in comparison is called maximum common subgraph of the two graphs if there is no other subgraph with more vertices. In [Bunke 1997], the authors introduce a graph matching technique based on an error-correcting strategy. They propose a cost function for a graph edit distance algorithm that allows to compute the maximum common subgraph between two graphs. They demonstrate that any other graph edit distance algorithm (with a suitable assignment of costs to edit operations) can be applied to compute maximum common subgraphs. Improvements to the graph edit distance measurement in [Bunke 1997] are proposed in [Fernandez 2001], based on [Valiente 1997]. Also, in [Messmer 2000], an algorithm for matching a single graph against a graph repository is presented. The algorithm, motivated in the ideas of the RETE algorithm [Forgy 1982][3] for rule matching in expert systems, bases on a recursive decomposition of the graphs in the repository into smaller subgraphs until reaching graphs of one vertex. The matching process uses the common parts of the decomposed graphs to avoid repeated comparisons with the query graph, resulting in a total matching time that has a sublinear dependency on the number of graphs in the repository.

A graph matching algorithm and accuracy metric for schema matching are proposed in [Melnik 2002]. The input to the algorithm are two graphs and the output is a mapping between corresponding vertices from the input graphs. Correspondence between vertices is defined by a matching goal and the results obtained by the algorithm are expected to be checked and adjusted by end users. To evaluate the accuracy of the algorithm, the authors propose a new accuracy metric (match accuracy). The metric is based on counting the number of needed adjustments to the algorithm results. The metric is adequate for problems such the schema matching, where the semantics associated to graph elements is a relevant challenge in the matching process.

[3]Based on earlier working material: Forgy, C., A network match routine for production systems (1974).

2.6.1.1 Pattern Matching in Graph Transformation Systems

In software engineering, graphs are often associated to models representing the architecture of software systems. Elements in architectural models often contain rich descriptions and, therefore, the type of graph representations should be sufficiently expressive (involving for instance graphs whose vertices and edges have types and attributes associated [Ehrig 2006a], [Heckel 2006]. A significant number of contributions involved with architecture modifications relate to the field of Graph Transformation (GT) systems [Rozenberg 1997]. Architecture modifications (or transformations) are also core to model-driven software development. For instance, in [Baresi 2006] an architectural style-based approach for SOA modelling and design is presented. Service architecture models are derived from refined business architectures. The refinement of a concrete business scenario is guided (automatically) by the refinement of an abstract business-level style into a service architecture style. Each refinement would provide for service architectures semantic correctness and platform consistency with business levels. The focus is on the ability of dynamic architecture reconfiguration where new services can bind at run-time.

Graph transformation rules in GT systems lead architecture modifications. For the single push out approach [Ehrig 2006a], a graph transformations rule consist of a left part (pattern graph) and a right part indicating the graph section that is to be transformed into the target graph. An additional intermediate graph is used in the double push out approach. A basic problem for GT systems is the problem of matching the pattern graph contained in the transformation rule with a subgraph in the target graph.

An algorithm to solve the graph pattern matching problem involved in the search of a redex for an arbitrary graph rewrite rule is introduced in [Zundorf 1996]. Redex is the subgraph in the target graph matched with the left side of the graph rewrite rule. The algorithm is implemented in a tool (PROGRESS) for the execution and implementation of graph grammar specifications. The aim is to improve the efficiency of a naive implementation of an algorithm for the graph matching problem (whose complexity is $O(N_L)$, with N the number of vertices in the target graph and L the maximum size of the left side of the rewriting rule). The algorithm is specially suitable for large target graphs and small subgraphs to be matched, for target graphs stored in data structures supporting efficient indexing schemes to access graph elements involved in the matching, and if a small number of graph rewriting rules is involved.

A number of GT systems implement different pattern matching strategies. Fujaba [Fujaba 2010] is an open source CASE tool aimed to support software forward and reverse engineering. Pattern matching in the context of GT originates from the tech-

nique presented in [Zundorf 1996], however the data model for graphs is updated to an object-oriented data model. AGG is a development environment for algebraic GT systems [Ermel 1999], [Taentzer 2004]. In AGG the matching of the left side of a rewrite rule may be partially defined by the user. Partial matches are automatically computed and several choices for completion can be chosen arbitrarily. All possible completions can be computed and shown one after the other in a graph editor. The (sub-)graph matching problem is solved as a constraint satisfaction problem, similar to the approach in VIATRA [Varro 2002].

An adaptive approach for graph pattern matching is presented in [Varro 2006a], where the optimal search plan can be selected from previously generated search plans at run-time based on statistical data collected from the current instance model under transformation. In [Varro 2006b], an approach to GT based on standard relational database management systems is presented. Graph rewrite rules are associated to database views, the graph (pattern) matching step is managed by inner join operations on tables that represent either a vertex or an edge of the rule graph, and negative application conditions are handled by left outer join operations. After executing inner join operations, the joined table is filtered by injectivity and edge constraints – injectivity constraints express the injective mapping of rule graph vertices and edges on the database level. Edge constraints define restrictions imposed by the graph structure. Then, a projection selects columns of the filtered joined table that represent vertex identifiers. Improvements to the efficiency of the pattern matching step are suggested for parallel processing and the use of built-in query optimiser features in database systems. The authors have carried out a performance comparison of different GT tools in [Varro 2005]. For the pattern matching step, the benchmark includes variations to the pattern size and maximum degree of out/in-going edges for vertices, with large patterns containing up to fifteen vertices and hundreds of out/in-going edges involved. The results indicate that the pattern structure and the appearance of negative application conditions also influence the performance of GT tools. The authors suggest that the improvement of GT tools should focus on developing more efficient techniques to process multiple matchings when a straightforward parallel matching approach is not possible.

GrGen is a generative programming system for GT that applies heuristic optimisations during the graph matching task. A notion of search plans to represent different matching strategies is used. Search plans involve a cost model that is optimised to select an adequate search plan. According to the authors, the performance of GrGen outperform the tools compared in the benchmark in [Varro 2005] for at least one order of magnitude. A search plan involves a sequence of search operations, with each operation representing the matching of a single vertex or edge of the pattern graph to an appropriate vertex or edge of the target graph. The whole

search plan describes the stepwise construction of possible matches between the pattern and target graphs. Finding a match fast depends on the chosen search plan. The selection of a search plan is based on an approach presented in [Dorr 1995a], [Dorr 1995b]. Lookup and extension operations are used to iteratively find matches (by type). Partly constructed matches are called candidates. Lookup operations add vertices or edges not connected to previously matched vertices. Extension operations add edges that are connected to previously matched vertices. Operations can cause the splitting of a candidate into several new candidates, which can lead to an exponential growth with worst case $O(PN^P)$ with P the number of vertices of the pattern, N the number of vertices in the target graph. If the execution of a search plan causes no splitting, a linear runtime for sparse target graphs $O(P)$ can be achieved.

2.6.2 Frequent Subgraph Discovery

Discovering patterns in large processes can assist the design of new reusable process-centric services. For process models represented as graphs, the problem of discovering a pattern can be considered a problem of discovering frequent subgraphs. A number of proposals for sequential and parallel calculation of the frequent subgraph mining problem to discover interesting patterns has attracted attention in diverse applications scenarios such as analysis in social networks, molecular compounds and document-based information retrieval [Han 2007], [Wang 1995], [Kuramochi 2005], [Bringmann 2008]. Solutions to the frequent subgraph discovery problem – such as the approximate solutions in [Kuramochi 2005] or the results of clustering-based approaches in [Jung 2006] – can be adapted to different process scenarios. However, according to [Greco 2005], the adaptation may require major efforts for moving to the process scenarios, hence dedicated solutions would be required.

The frequent subgraph discovery problem is addressed in two main settings. The graph-transaction setting refers to the discovery of subgraphs that occur frequently across a set of target graphs (graphs in a repository). The single-graph setting refers to the discovery of subgraphs that occur multiple times in a single large input graph. For the single-graph setting scenario [Kuramochi 2005] propose algorithms to obtain approximate solutions. The algorithms have their origin in algorithms developed for finding frequent itemsets and sequences [Agrawal 1994]. The subgraph isomorphism problem (involving a one-to-one mapping) is a derived problem addressed in [Kuramochi 2005]. This problem, as indicated previously, can cause a significant performance decrease in the algorithms. For large sparse graphs (in the single-graph setting scenario) the authors indicate that algorithms have a good performance when finding subgraphs that have many edge-disjoint embeddings. Large sparse graphs are typical for process descriptions [Golani 2003] and frequent process substructures

are expected to be edge-disjoint. Possible adaptations of the two proposed algorithms in [Kuramochi 2005] to a scenario of a large and complex graph-based process description would require significant efforts regarding the method for canonical labelling, which provides a unique code (label) to each graph in a repository and it is used to check whether two subgraphs are identical or not. Difficulties can arise due to the inherent complexity of the descriptions for process elements. Similar to [Kuramochi 2005], the advantages of the solution proposed in [Inokuchi 2005] for mining frequent subgraphs in labelled graphs relies on the algebraic representation of graphs and its organisation to limit the search space efficiently.

A solution to the problem of identifying frequent patterns of workflow executions is proposed in [Greco 2005]. The proposal focuses is on discovering the most frequent substructures (patterns) of workflow executions. Workflow schemas and their occurrences are formalised as acyclic graphs. Future extensions involve the representation of cycles. Comparative analysis of realistic process execution data are also required. In [Greco 2008], the same authors propose a mechanism for mining taxonomies of process models. In this case, the focus is on extending process discovery mechanisms [Cook 1995] to a method producing a taxonomy of workflow models. Models closer to the root of the taxonomy are more abstract and they are used as abstractions during post-workflow executions analysis. Future improvements are suggested, such as enhancing match functions to consider the semantics of activity descriptions and further attributes like usage statistics and performance metrics in order to enable a more adequate clustering of activities.

2.7 Summary

This chapter had described a number of contributions in the literature with regard to SOA methodologies, service identification strategies and the use of architecture and model abstractions for SOA design. Existing SOA methodologies (see Section 2.3.1) have advanced the way services and their organisation are designed and developed; however, there are still challenges such as providing integration between modelling views, guidelines and tool support to design services with adequate granularity and satisfying the constraints imposed by existing applications and mediums to reuse knowledge embedded in architecture and model abstractions documentation.

This work proposes a framework to assist the design of service-based systems for processes and applications integration. The framework integrates business process, service architecture and application architecture layers in an integrated modelling environment with explicit traceability support, and its distinguishing characteristic is that it uses patterns across layers to assist the design of services. In this work, *patterns* refer to a set of constraints over elements and their relationships in concrete models,

including architectures, and they capture known solutions to recurring problems.

Processes are central to the addressed integration problem, hence *process* patterns are essential to the proposed pattern-based framework. Patterns at software level have been widely studied and exploited within the software community; however, only recently have patterns and their use at a more business operation level received more attention. A number of articles proposing different notions of process patterns were discussed. The concept of activity pattern based on recurrent functions in [Thom 2009] and actions pattern in [Smirnov 2009] are close to the concept of *process pattern* in this work. Similar to software level patterns, process patterns define constraints over process elements and their relationships in concrete models. They abstract recurrent (process) steps across processes and organisations and; therefore, they represent an opportunity to guide the design of reusable process-centric services. In a similar intention to [Papazoglou 2006a], [Papazoglou 2006b], where the authors suggest the use of reference models and associated business-level abstractions to guide the definition of normalised business functions for services, this work uses process patterns identified in process level descriptions to guide the design of services for integration systems. The proposed approach aims to advance techniques for pattern-based service definition by providing an automatic mechanism for process pattern identification.

Service architectures can be designed using existing services, but often enterprise SOA initiatives involve the design of new services. Section 2.4 described a number of approaches for identification of new and existing services. Ideally, new designed services can become services that are actually reused across processes in organisations. Pattern-based service design approaches focus on this principle of reuse. However, they need to tackle a practical challenge. Instances of process patterns have to be identified in often large and complex (process) models. Patterns could be known patterns, but also unknown patterns (i.e., there may be recurrent process sections in a process model, but the analyst or architect does not know that they exist). Pattern matching and pattern discovery techniques can be used to identify known and unknown patterns in process models. A number of these techniques are graph-based techniques. These techniques can be also used to identify existing services from repositories of process-centric service descriptions. Sections 2.4.2 and 2.4.1 described a number of these approaches. They assist and automate some steps, but they have a number of shortcomings including the significant processing time that they may need to obtain results, the matching or similarity calculation could be only based on the syntax of process and service descriptions, and they may be inaccurate when partial or inexact matching is involved. In subsequent chapters, the proposed pattern matching and discovery techniques shall be described and evaluated. Without going into detail, it can be indicated that the proposed graph-based pattern matching

technique is fairly efficient for structural matching steps and the overall efficiency would depend on the complexity of the adopted approach for semantic matching of vertices descriptors. Towards the end of this chapter, some problems with exact and inexact matching and how the proposed approach addresses them are discussed.

Pattern matching and discovery techniques have been also proposed for software level patterns in the context of software maintenance and reverse engineering approaches (Section 2.3.3), however they do not apply directly to process pattern matching and discovery, since process elements and their relationships have particular semantic and behavioural characteristics. Additionally, pattern matching in the context of model transformation systems has also been discussed. Although they are frequently associated with architecture model transformations – where pattern matching is rather involved with software levels – they could be also adapted for process models. However, if pattern instances are identified in large graph-based process models, a number of pattern matching techniques designed to work in a graph-transaction setting (involving the matching of patterns in a repository of small/medium size graphs) and benefiting from canonical labelling or efficient indexing methods may require significant changes to be adapted to the single-graph setting.

Process models or process-centric service descriptions are not always available. In Section 2.5, process mining techniques were mentioned. Even though the emphasis of these techniques is different from matching and discovering patterns, they attempt to obtain process models that best represent sets of process executions. In that sense, process mining techniques are an input to the addressed problem, providing the models from where patterns could be matched or discovered.

Figure 2.1 shows a diagram summarising the relations between the proposed framework and techniques, and theoretical foundations, background research and related work.

Some Problems in Exact and Inexact Graph Matching. In the field of pattern recovery and process model comparison, some problems with exact and inexact graph matching have been discussed. As pointed out in [Tsantalis 2006], in practice, inexact pattern matching techniques to identify patterns in models, which are based on graph edit distance algorithms, can generate inaccurate results. Take the example at the top of Figure 2.2 that shows the segment of two software systems and a design pattern. A technique relying of graph edit distance would identify an instance in System 2 as closer to the pattern definition because there are fewer edit operations to transform the system graph into the pattern graph, even though System 1 has the pattern completely instantiated.

Different workflow similarity measures used in service discovery techniques are analysed in [Wombacher 2006]. Similarity between workflows (processes) focus-

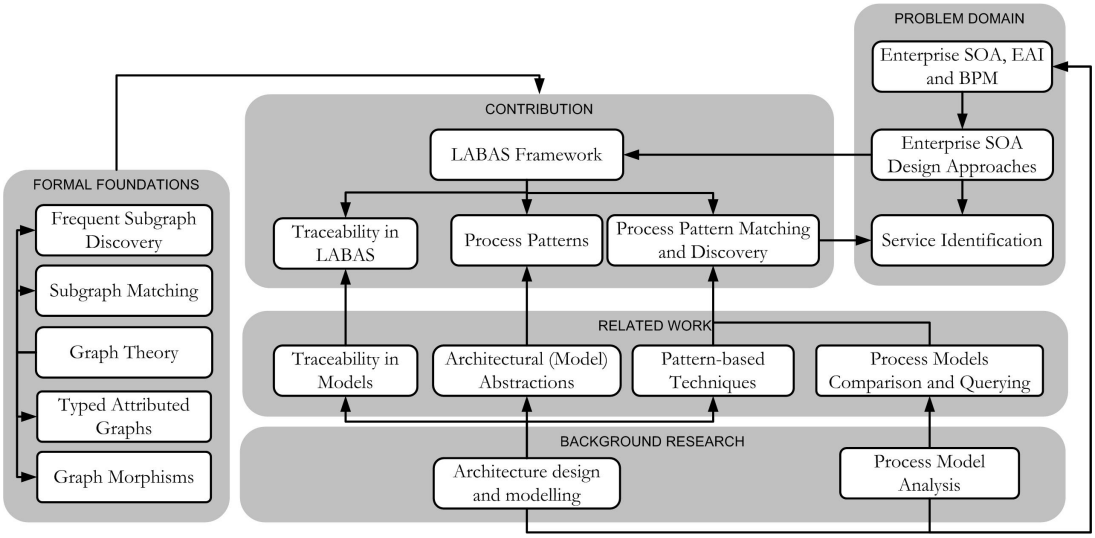
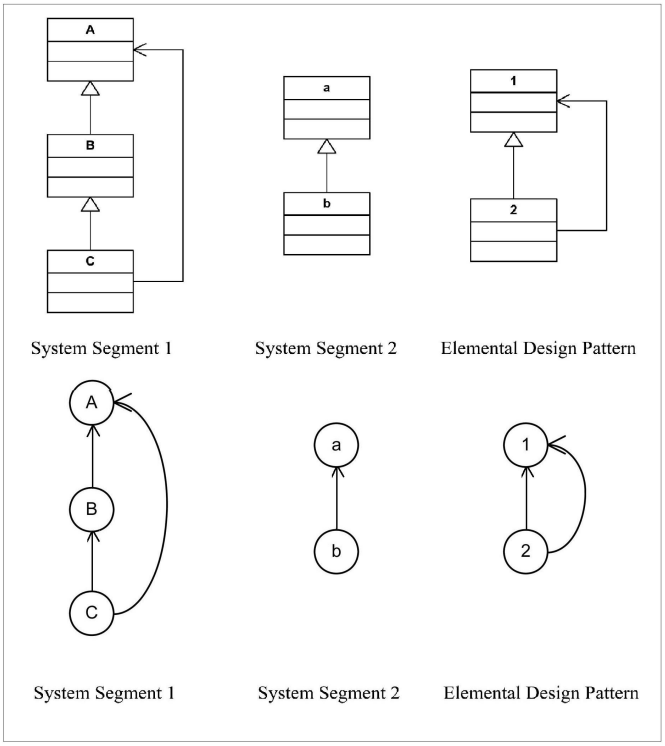


Figure 2.1: Relation between the proposed framework/techniques and theoretical foundations, background research and related work.

ing on behaviour and processes are represented as annotated finite state automata, which allow synchronous communication. Beyond possible shortcomings regarding computational complexity when using automata to represent processes, the authors in [Wombacher 2006] describe other challenges for techniques calculating similarities between processes (represented as automata in this case). Take the example at the bottom of Figure 2.2 where twelve automata are depicted. The structures of automata 1, 2 and 3 are different, however their represented behaviours are the same. Since the structures are different, techniques relying on isomorphic structural matching to compare automata can provide inaccurate results. The criteria that the authors indicate for similarity measure are the following. (C1) The similarity measures between each combination of automata 1, 2, and 3 must be the same. (C2) The similarity measures of one out of automata 1, 2, or 3 and automata 4 and 5 respectively are the same. (C3) The similarity measures of two out of automata 4, 5, and 6 are the same. (C4) The similarity measure of automata 7 and 8 must be higher than the similarity measure of automata 7 and 9. (C5) The similarity measure of automata 7 and 9 must be higher than the similarity measure of automata 7 and 11.

To finalise this review of the state of the art, the results that the proposed techniques in this work would provide with regard to the criteria in [Wombacher 2006] and problems illustrated in [Tsantalis 2006] are shown using the examples in Figure 2.2. The results provide an indication that the proposed techniques satisfy the criteria in [Wombacher 2006] (Figures 2.4, 2.5) and address the problem indicated in [Tsantalis 2006] that several other techniques are lacking (Figure 2.3). Also, even though the particular representation of processes explained in the next chapters is



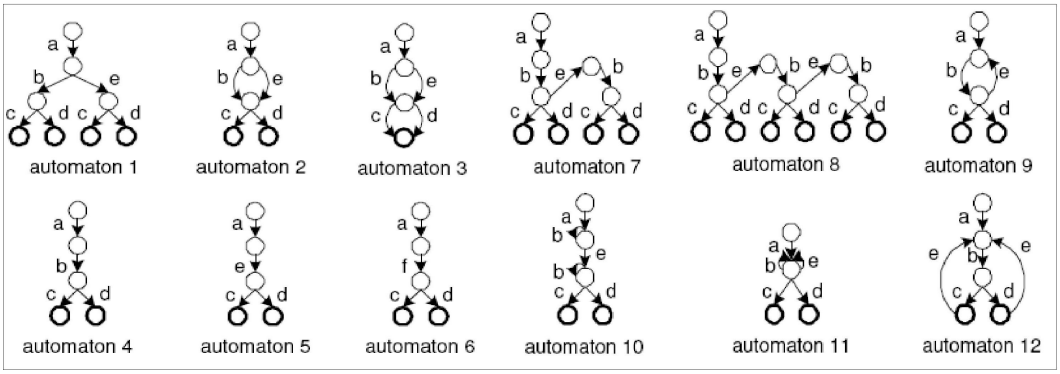
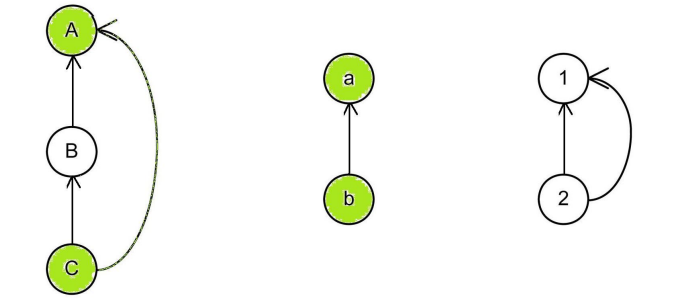


Figure 2.2: Examples to illustrate problems in exact and inexact matching. Figures adopted from [Tsantalis 2006] and [Wombacher 2006].

different to an automata-based representation, the techniques proposed apply to any graph-based representation, including automata.

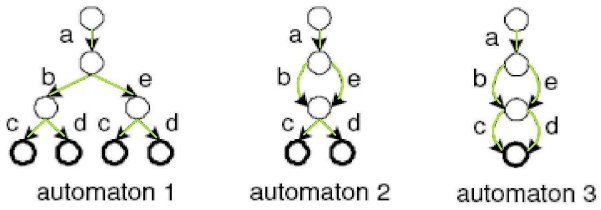


For A, C and (C,A) : Match score = 3 / (exact / partial)
For B, (C,B), (B,A) : Match score = 0 / (exact / partial)

For A, C and (C,A) : Match score = 3 / (inexact / complete)
For B, (C,B), (B,A) : Match score = 3 / (inexact / complete)

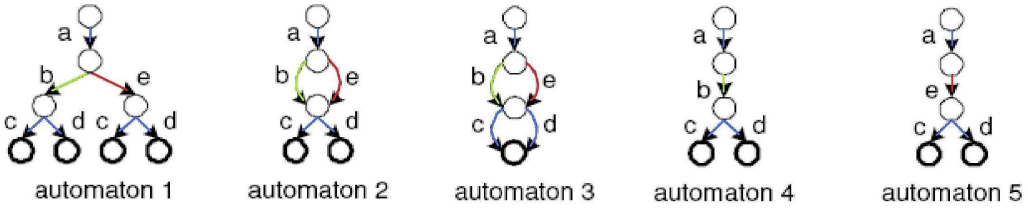
For a, b, (a,b) : Match score = 3 / (exact / partial)

Figure 2.3: Results that proposed techniques would provide for cases in Figure 2.2.



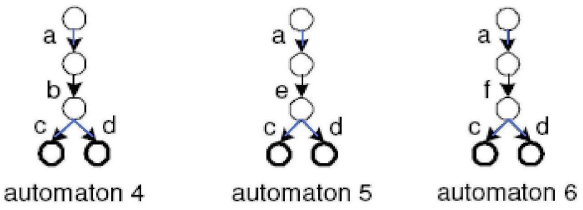
C1: The similarity measures between each combination of automaton 1, 2, and 3 must be the same.

Directed Edge-based exact/partial match
Match(a2,a1) = 5 3 one-to-one, 2 one-to-two matches
Match(a3,a1) = 5 3 one-to-one, 2 one-to-two matches
Match(a2,a3) = 5 5 one-to-one matches



C2: The similarity measures of one out of automaton 1, 2, or 3 and automaton 4 and 5 respectively are the same.

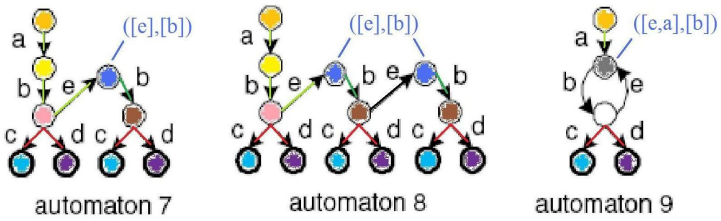
Directed Edge-based exact/partial match
Match(a4,a1) = 4 2 one-to-one, 2 one-to-two matches
Match(a4,a2) = 4 4 one-to-one matches
Match(a4,a3) = 4 4 one-to-one matches
Match(a5,a1) = 4 2 one-to-one, 2 one-to-two matches
Match(a5,a2) = 4 4 one-to-one matches
Match(a5,a3) = 4 4 one-to-one matches



C3: The similarity measures of two out of automata 4, 5, and 6 are the same.

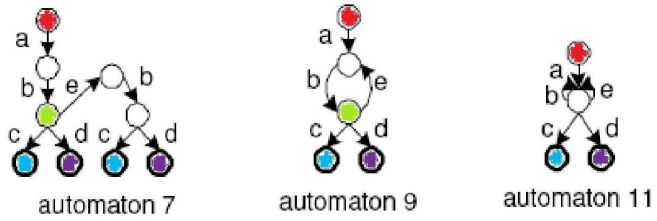
Directed Edge-based exact/partial match
Match(a6,a4) = 3 3 one-to-one matches
Match(a6,a5) = 3 3 one-to-one matches

Figure 2.4: Part-A: Results that proposed techniques would provide for cases in Figure 2.2.



C4: The similarity measure of automata 7 and 8 must be higher than the similarity measure of automata 7 and 9.

Directed edge- and vertex-based matching
Match(a7,a8) = 6 vertex matches (3 one-to-one,
2 one-to-two, 2 one-to-three matches)
+ 6 edge matches (3 one-to-one,
1 one-to-two, 2 one-to-three matches)
Match(a7,a8) = 3 vertex matches (1 one-to-one,
2 one-to-two) + 3 edge matches
(1 one-to-one, 2 one-to-two)



C5: The similarity measure of automata 7 and 9 must be higher than the similarity measure of automata 7 and 11.

Directed edge- and vertex-based matching

Match(a9,a7) = 4 vertex matches (2 one-to-one,
2 one-to-two)
Match(a11,a7) = 3 vertex matches (1 one-to-one,
2 one-to-two)

Figure 2.5: Part-B: Results that proposed techniques would provide for cases in Figure 2.2.

Chapter 3

A Framework for Processes and Applications Integration

Contents

3.1 Motivation

Enterprise Application Integration (EAI) aims to link separate *applications* into an integrated system driven by *business* models and the goals they implement. A central problem of application integration is maintaining alignment between the business and technical dimensions involved. Business processes do not map one-to-one to service architecture processes. This gap has turned out to be difficult to approach systematically and to automate [Erl 2004]. Architecture abstractions like patterns and styles capture design knowledge and allow the reuse of successfully applied designs, thus benefitting the quality of software. Abstraction is a central driver in

software engineering approaches. At the business (modelling) level the reuse of successful designs is equally important.

The development of integrated enterprise-wide application architectures is a continuous process. To improve the process and overall quality, the experience of analysts, architects and developers should be captured and reused. Knowledge gained from integration projects can be captured to build a repository of experience-based pattern solutions. A coherence framework able to capture different architecture-level descriptions and abstractions (such as patterns) into an integrated view with business and software perspectives is required. The problem of process and application integration in terms of modelling aspects, but also architecture issues involve:

- modelling aspects – modelling different views such as information hierarchy, behaviour, and semantics in a coherent framework with interrelated models.
- architecture aspects – architecture description and abstractions such as patterns are central. To benefit from reusable architecture design knowledge captured in the form of patterns, pattern identification and utilisation are basic activities requiring support.

Note that vertical transformations – from business to software architectures – are also important. Elements involved with business operations should be mapped to services. Synchronisation between the two modelling layers is required. Although this work does not address this aspect, aiming at completeness, a strategy for pattern-based vertical transformation is introduced in the context of the proposed framework.

The rest of this chapter presents an architectural framework to address the problem of enterprise process and application integration. A layered architecture for service-centric process and application integration is the backbone of the proposed framework. Section 3.2 presents the modelling framework, its layers and involved models. It also describes the role of patterns in the framework, how they can be described by end users and utilised within the framework. Section 3.3 describes the pattern-based techniques that can help analysts and architects during services and service architecture design. Among the techniques, pattern-based service identification (pattern matching and discovery) is developed in detail in this work, but described in the next chapters. Section 3.4 describes a traceability-based approach used to integrate the different modelling layers, including a higher abstraction layer containing patterns. The utility of this approach for change impact analysis is discussed.

3.2 Layered Architecture for Business, Applications and Services

This section presents a framework that integrates different perspectives of the process and application integration problem. It captures process and architectural models and patterns in an integrated structure. The framework is organised in a layered architecture called LABAS – Layered Architecture for Business, Applications and Services. It layers separate aspects and aid with the maintainability of the architecture [Bass 2004].

Two interrelated perspectives of the integration problem, the business and application view, and two aspect of the solution, services and patterns, are considered in LABAS:

- Business view – two main models describe the business dimension of the EAI problem: process models and domain models. While business process models capture the dynamics of the business, domain models capture structural relations between business concepts.

- Applications view – an application architecture represents a system composed of several software applications. Application architectures at enterprise scale normally grow in a decentralised way and involve different technologies, paradigms of development and modelling notations.

- Services view – services organised in an architecture have the potential to bridge the gap between business and technology and to improve the reuse of existing applications and the interoperability with new ones. Software services are the building blocks of service architectures. They can be composed to provide more complex functionality and to automate business processes.

- Patterns – reusing proven solutions reduces costs and development time and ensures coherently integrated and architected application systems aligned with business processes. Using patterns enables architects to implement successful application integration solutions. Patterns are core and a distinguishing characteristic in the LABAS framework.

Moreover, change management and traceability are important aspects of IT systems and business alignment. Consistence between LABAS layers is enhanced through explicitly connecting modelling elements of the EAI problem with elements of the service architecture solution. In order to provide traceability support to elements involved in the integration problem across layers, LABAS provides explicit

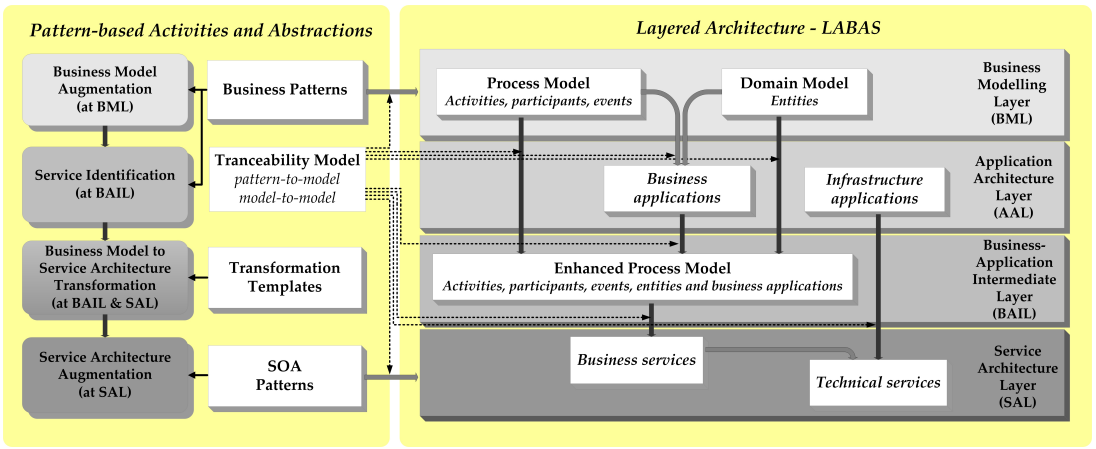


Figure 3.1: Layered Architecture for Business, Applications and Services (LABAS).

links between elements from different layers. An explicit traceability model maintains the dependencies between elements of the integration problem and the service-based solution.

3.2.1 Layers in LABAS

The different layers organised in the LABAS framework capture the *problem* perspective, considering business and application architecture elements, and the *solution* perspective, involving elements of the service-based integration solution. From a modelling point of view, a meta-model defines the common constructs among the different layers and provide the modelling support for the transformation from business to software levels. Figure 3.1 shows the layers in LABAS.

- Business Modelling Layer. The Business Modelling Layer (BML) represents the business context of the integration problem. BML is a container for elements of the process and domain models. A concrete notation (BPMN [OMG 2008b]) is utilised to implement process modelling support at BML. Process models captures events, activities/tasks and roles involved in actual business processes. The main focus is on behaviour. On the other hand, domain models provide a structural view of business concepts and their relations. Domain modelling is supported through standard UML [OMG 2007] support. A basic ontology to represent domain information model entities can provide extended facilities [Gacitua-Decar 2009b].

- Application Architecture Layer. The Application Architecture Layer (AAL) is a container for the application components supporting the business processes in BML. AAL is organised as a process-wide application architecture. Thus, ap-

plications can support different tasks and be owned by different roles defined in process models at BML. In order to describe an AAL model in architectural terms, the component and connector view from [Garlan 2006] is adopted. It describes a software system as a set of components, where each component has a set of ports with interfaces that enable the interaction with other components through connectors.

- Business-Application Intermediate Layer. The Business-Application Intermediate Layer (BAIL) provides a consolidated view of the integration problem. The aim behind jointly modelling the business and applications views is to derive an integration solution from BAIL models. The consolidated model integrates the models through trace links (see more details in Section 3.4). The integrated models are the business process model, the domain model and the application architecture model. Trace links between process model and domain model elements, and between domain model elements and application components, are the core to consolidate the the integration problem covering elements of the business and application views at BAIL.

- Service Architecture Layer. The Service Architecture Layer (SAL) provides a service-based integration view. It is a container for software services. These services, organised in a service architecture [Alonso 2004] implement the integration solution. A process-centric component and connector view [Allen 1997] to represent AAL elements was used and it follows the recommendations of the Object Management Group (OMG) for service-oriented architecture modelling [OMG 2009b]. Additionally, the LABAS framework proposes a pattern-based service identification technique (see more details in Section 3.3) to support the design of service-based solutions for enterprise process and application integration problems. This technique is based on solutions for graph matching and motif discovery which are formalised and described in more details in Chapters 5 and 6.

3.2.1.1 Abstract Syntax for Models in LABAS

Concrete models in the LABAS framework require a modelling infrastructure that defines the abstract syntax to create specific models and trace links connecting models and architecture abstractions. Model-based design provide concepts and techniques to define a syntax to create concrete models. A metamodel defines the modelling constructs and its relations which are used to create specific models. The Object Management Group (OMG) is a large standardisation body addressing, among other aspects, the development of modelling standards and guidelines for model-

driven development. Modelling requirements for layers in LABAS can exploit metamodels defined by this organisation. Section 3.2.1 indicated the type of models considered in each LABAS layers. The abstract syntax used for these models is as follow.

- In the BML layer, process model elements follow the recommendations of the BPDM (Business Process Definition Metamodel) [OMG 2008a]. Domain model elements and their organisation follow the UML specifications [OMG 2007] for class diagrams. Only static aspect of classes are considered for domain models in BML.
- In the AAL layer, application architecture models and their elements follow the specifications for UML component diagrams.
- In the SAL layer, service architecture models use modelling constructs recommended in the OMG's UPMS (UML Profile and Metamodel for Services) [OMG 2006].
- A *traceability* metamodel adds to the LABAS modelling infrastructure facilities to manage explicit trace links between elements in different layers and between model elements and pattern elements. More details are provided in Section 3.4.

A LABAS *profile* was developed to create concrete LABAS's models using a standard UML tool. Section 7.3.1 refer to profile used in the proposed framework and support for pattern documentation.

3.2.2 Patterns in LABAS

Architectural abstractions such as patterns have diverse definitions in the software and business communities. Section 2.3.2 refers to different notions of architectural abstractions, in particular, Section 2.3.2.2 focuses on diverse definitions for process-oriented patterns. Figure 3.2 illustrates different levels of abstraction where the pattern concept can be positioned. At the concrete pattern level, pattern users can document patterns associated to a specific application domain and utilise them in that context. The focus in subsequent chapters is on process patterns at the concrete pattern level. Once a pattern is selected at this level, it can be instantiated in a concrete model, i.e., at the model level. More abstract patterns, which are independent of the application domain are expressed with an abstract syntax that is located at a metamodel level. Examples of process patterns in the abstract level are the process patterns defined in [WP 1999], [Aalst 2009b], [Weber 2008], [Lanz 2009]. They define control flow constraints for executable processes, however they are independent of the applications where they are involved in. Only recently, except for a few exceptions, such as in [Malone 2003], [Barros 2007], [Thom 2009], [Smirnov 2009],

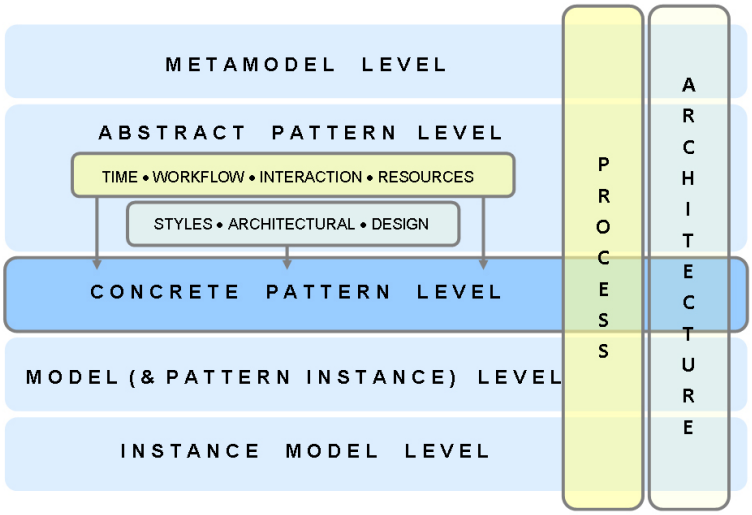


Figure 3.2: Abstraction levels and types of patterns.

knowledge from process designs created for common business operations is being documented and reutilized in a similar way to design patterns in software engineering. In this work, process patterns are addressed in this sense, i.e., in an analogous manner to patterns for software architectures.

The authors in [Buschmann 2007] suggest that the use of (software) patterns is not possible to be fully tool-based or automated. Although the proposed LABAS framework considers the notion of patterns similarly to the authors in [Buschmann 2007], this work emphasises on *automation* and therefore, pattern descriptions that make machine readability possible are encouraged. The aim is to help pattern users to automate tasks which are repetitive, time consuming and difficult to manage due to size complexity.

Subsequent chapters explain how the automated use of patterns in the context of LABAS is explored, in particular, the application domain of patterns in this work is focused on the business process level. However the techniques developed in next chapters can be applicable to any graph-based representation of software level patterns. The reason to focus on process patterns at business level is that design and architectural patterns at software level have been widely studied and exploited within the software community, however patterns and their use at a more business operation level are less investigated. Moreover, since service-based architectures are closely related to both the business operation and software perspectives, the LABAS framework should not only look at patterns from the software development perspective, but also from the business process design and integration perspective.

Business- and architecture-level patterns are the two main kinds of patterns distinguished in LABAS. *Process patterns* falling in the first category are central in the

next chapters:

- *Business-level patterns* identify the interaction and structure between users, business processes and data. *Process-oriented patterns* within this category capture common process designs providing a solution to frequent problems associated to business operations. Thus, not only structure but also behaviour are central to process-oriented patterns.

- *Architecture-level patterns* – at a more technical level – address enterprise application integration problems and capture reliable architecture solutions. In contrast to business-level patterns, architecture-level patterns shift the focus into more structural software component and service connectivity aspects. They can also have a direct link to quality attributes.

Process Pattern. In particular, this work associates a business *process pattern* to a description of a reoccurring design problem with regard to an organisation's operation in a specific business domain, and a generic process representation explaining a solution to the operational problem. Beside the problem-solution pair description, the constraints to the problem and the solution space are also explained. A generic solution to a process-centric problem is specified describing the constituent elements of the solution process, their responsibilities and relations. Processes are mainly focused on behaviour and; therefore, relations in a process pattern description are mainly associated to that aspect. A more formal (and graph-based) definition for process patterns is provided in Chapter 4.

3.2.3 Pattern Description for End Users

One important aspect facilitating the use of patterns is how they are described. There are several approaches to describe patterns, from natural language and more informal graphical representations such as the pattern descriptions in [Buschmann 2007], to more formal descriptions such as those based on role concepts [Kim 2008], graphs [Bottoni 2009] or with an ontological base [Pahl 2007] and [Kampffmeyer 2007]. The first type of descriptions are easily understood by humans but hardly readable for machines. More formal and precise descriptions benefit automation, but they might be difficult to use among pattern users.

A common and widely accepted medium to describe patterns is by means of *pattern templates*. A pattern template is a predefined structure to organise the main elements describing a pattern. This structure often involves an evocative pattern name, a concrete and precise description of the context, problem statement, constraints limiting the solution space and a solution that solves the problem within the

constrained solution space. Additional elements of a pattern template can include related patterns, pattern variants and examples where the pattern has been applied.

LABAS is a framework oriented to support the design of service-based solutions for enterprise process and application integration. Regarding the use of patterns in LABAS, a balance between facility of use among end users and formalisation to facilitate automation is considered. Thus, patterns are described using a *pattern template*. The template has parts oriented towards pattern users and others designated to be more easily processes by machines. The sections oriented to pattern users involve textual descriptions that use a vocabulary familiar to pattern users. As emphasised in [Rising 2007], a vocabulary and concepts familiar to pattern users are key to facilitate the use of patterns. On the other hand, one of the aims of this work is to automate some tasks related to the use of patterns. Automation is facilitated by providing a section in the pattern template – referred to as *pattern configuration* – that contains a graph-based representation of the pattern elements' configuration.

3.2.3.1 Pattern Template

Pattern templates in LABAS organise the set of patterns used during the design of service-based solutions for enterprise application and process integration. A pattern template includes the following sections:

Name. The name should be recognizable and useful to facilitate its searching and communication among pattern users. The name provides a reference to users familiar with the pattern to immediately bring to mind the problem and solution addressed by the pattern.

Problem. The problem section describes a recurrent design problem addressed by the pattern. A description of the problem also contains its *context*. The problem description and its context have a textual explanation targeting pattern users. The pattern *context* might contain a machine readable section. That section uses a predefined categorisation of the specific domain addressed in the problem. Constraints are described for concepts in that domain. Categories and constraints can be defined in a domain specific ontology that helps to divide, organise and explain the problem space and support automation during the utilisation of patterns.

- *Context*. The context defines a set of conditions in which the pattern problem occurs. It can be thought as a constrained problem space, similar to the constrained solution space described by the *Forces* section in the *Solution*. In one extreme, a problem space without constraints cover an universal problem. This extreme is not desirable since, as explained in [Buschmann 2003], a pat-

tern can become all things for all people, each person with their different – perhaps incompatible – view, potentially damaging pattern utilization. The more constraints are defined for the problem space, the more precise the context description. However, if the context is significantly restricted, the problem addressed by a pattern can lose recurrence and; therefore, dilute the utility of the pattern.

Solution. The solution identifies the *configuration* of elements balancing the *constraints* within the *context* of the problem. Elements of a pattern solution are generic and often named *pattern roles*. Simple pattern role descriptions are composed of an evocative name, a textual description for pattern users and possibly a category from a domain specific ontology. If a category is not assigned, the pattern role name is used as a more informal medium of identification.

Pattern roles are connected to each other in a *pattern configuration*. Connections represent either static or dynamic relations among pattern roles. Comprehensive pattern role descriptions contain a set of attributes describing them. In order to support automated mechanisms to compare pattern roles, a threshold value can be assigned to each attribute. This value represents a quantitative indication for the minimum similarity value between two attributes being compared. Section 5.5 explains the use of these attributes and threshold values in more details.

Specific types of pattern involve different types of pattern roles. Architecture patterns in LABAS consider the component and connector view from [Garlan 2006] to represent pattern configurations. Components, connectors, interfaces and ports are the types involved. Process patterns on the other hand refer to configurations that relate pattern roles from a control flow perspective, reflecting the abstract behaviour of a number of process instances. Process participants (or process pattern roles) are involved with or in activities, decisions, events and domain entities that together define a process pattern configuration.

- *Forces*. Forces define the influential factors restricting the possible solutions to a pattern problem. They describe the *constraints* limiting the solution space associated to a pattern problem. The information that forces provide help to pattern users to select an adequate solution for particular problems. Similar to the *Context* section, the *Forces* section describes the constrained solution space of a pattern description. Note that a solution space which is significantly restricted could make a pattern specific to such a degree that it can become no more than a particular design solution but not a pattern.

Consequences. The consequences of applying a pattern in a determined context are described to facilitate a reasoned design decision. If a pattern is applied, what

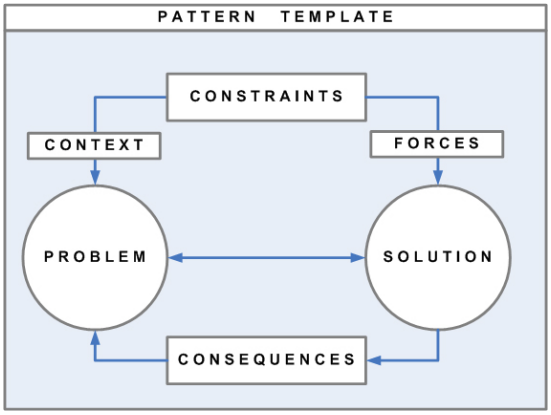


Figure 3.3: Relations between elements of a pattern template.

should a designer expect? A textual description is included in this section of the pattern template. The consequences section can also provide a reference to effects on quality attributes after applying a determined pattern to a design [Babar 2004], [Harrison 2007].

A schema of how the different parts of a pattern template are related is depicted in Figure 3.3. It illustrates the symmetry of the *problem-solution* pair, its restrictions through the *context* and *forces* and their *consequences*. After a solution indicated by a pattern is applied, consequences can create new problems.

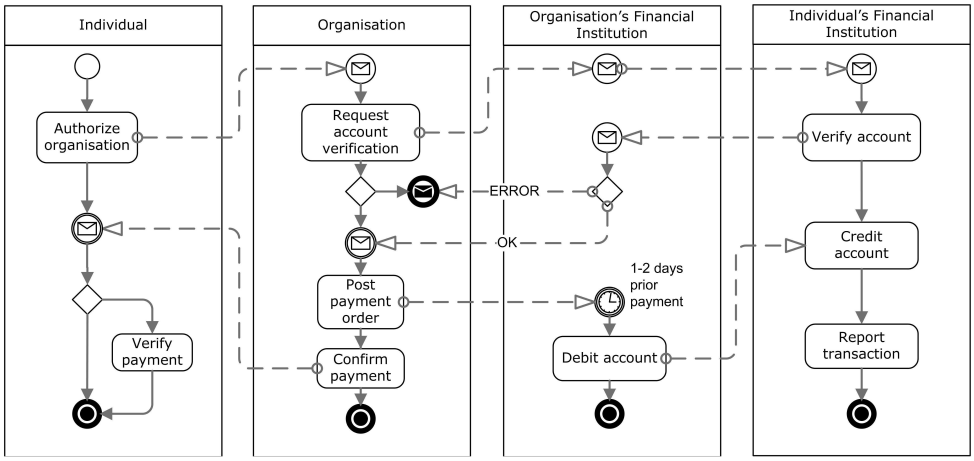
Example. A simple example is used to illustrate the use of a pattern template. The example mostly describes the information provided to pattern users. Machine processable information used in this work can be derived from the pattern configuration section. A graph-based representation of the pattern configuration can be derived an processes by algorithms proposed in next chapters. The intention of the example here is to illustrate what pattern users can expect from a pattern description in a pattern template.

Suppose a company is dealing with the problem of reducing costs associated with issuing and settling physical bills for expense and travel reimbursements. The company is also interested in providing a quicker and automatic way to reimburse bills to its employees and avoid the use of paper checks. One alternative solution is considering an electronic bill presentment and payment scheme. The Electronic Payments Association [NACHA 2010], a large standardisation body, suggested a number of generic solutions to the mentioned problem. These generic solutions are associated to constraints imposed on specific interactions between roles involved in the bill presentment and payment process. The most basic type of payment is a Direct Deposit [EPN 2010], used for payroll, expense and travel reimbursement, pension and annu-

ity payments, among many other concrete types of payments. Direct Deposit can be considered as a process-oriented pattern at the business level.

Table 3.1: *Domestic direct deposit* pattern template

PATTERN TEMPLATE SAMPLE	
Name	Domestic Direct Deposit
Problem	An organisation needs to reduce costs associated with issuing and settling physical bills and automate the payment process. The organisation wants to avoid the use of paper or other physical payment documents.
Context	Payments are not regular but they occur eventually. Individuals receive physical bills and they need to process manually their payment documents, for example, by means of cashing paper checks. All employees have a bank account. Financial institutions associated to employee's and the organisation's bank accounts are in the same country. Organisation are of any size.
Solution	An individual authorises an organisation to make a *direct deposit* into one or more of its accounts. Once an authorisation is processed, the organisation making the direct deposit may perform a test run with no amount to make sure the account details have been recorded correctly. If the account information is correct, the organisation processes the direct deposit transaction by delivering or transmitting a special file to the individual's financial institution. The organisation's financial institution processes the direct deposit file through the network of financial institutions one or two days prior to the payday or payment date. On or before the direct deposit date, the organisation's financial institution debits the individual's account for the total amount of the direct deposit transaction that were on the direct deposit transaction file. The financial institutions that receive the direct deposit transaction credit the individuals account and report the transaction on their account statements. The organisation should provide a statement of the direct deposit payment.
Pattern roles	Individual, organisation, individual's financial institution, organisation's financial institution.
Pattern configuration	



Forces	Authorization forms can vary from organisation to organisation, but the required information should be the same. Information of individual's accounts should be confidential. This can increase the effort to secure personal information. Electronic direct deposits can be domestic and international. Domestic direct deposits involve less organisations. The network of financial institutions requires between one or two days to process a transaction debiting from one account in a financial institution and crediting to other account in a different financial institution. This is true when both institutions are in the same country, if they are in different countries the processing time can be increased. Each country has their own processing requirements. If a pay date falls on a holiday or other dates when financial institutions are closed, the amount of money should be available on the day before the scheduled pay date. In the case of a direct deposit of payroll, the statement should in general include the same information provided on a paycheck stub, including deductions and the net pay amount. An individual may have the option of splitting a deposit or distributed among different accounts even from different financial institutions. In the former case, all activities after to the payment order posting are repeated for each split deposit. In the latter case, all activities referred in the pattern configuration are performed for each account in each financial institution.
Consequences	Payment process is automated and physical payment documents eliminated. Depending on the number of individuals, the costs associated with issuing and settling physical bills would be reduced.

3.3 Pattern-based Techniques

A service-based solution to the integration problem is incrementally obtained by transforming models at business level and their restrictions at application level to a service architecture solution. The transformation is supported by patterns and techniques that facilitate their use. Pattern-based techniques provide support to business

analysts and software architects to incrementally design a service architecture solution. The main activities involving the use of patterns in the LABAS framework are: business model augmentation, services identification, business model to service architecture transformation and service architecture augmentation. Note that two techniques, pattern matching and discovery for service identification, are developed in this work. The other pattern-based techniques are only introduced in an attempt to define the overall environment for the proposed pattern-based approach to SOA design. Figure 3.1 indicate the relation between activities and layers in LABAS.

3.3.1 Business model augmentation

This activity involves the intentional application of one or more process patterns to a business process model. The result is an augmented process model. This model has one or more instances of a single or several process patterns. A number of tasks can be performed prior to the instantiation of a pattern: patterns can be *recommended* before their instantiation, they can be *compared* and *modified* to accommodate their appropriate utilisation. After a particular pattern is selected, it can be *instantiated*. If several patterns are being instantiated they might be *combined* to cover a broader design issue. Techniques and formalisation can facilitate automation during business model augmentation.

- *Pattern recommendation.* Less experienced designers (business analysts, software architects, developers) might not be aware that a pattern can be applied to improve a particular design or solve a design problem in that context. The use of a pattern is often triggered by the recognition of similarities between the problem addressed in the pattern and the problem faced by the designer. Automated pattern recommendation requires that the description of the problem addressed by a pattern is expressed in a way that machines can process [Kim 2007]. Pattern descriptions containing machine processable problem-solution descriptions can be considered as more feasible candidates for automated pattern recommendation. A problem can be systematically searched in a model, localised and subsequently a solution can be recommended to end users.

- *Pattern comparison.* A design problem can be addressed by more than one pattern and therefore there might be more than one solution. Two or more competing solutions require comparison. Competing solutions can be analysed considering the information provided in the consequences section of pattern templates. One example is to consider the effects of patterns on particular quality attributes [Harrison 2007], [Babar 2004].

- *Pattern modification.* A pattern defines a generic solution. The application of a pattern to a particular model (design) often requires to adjust the generic solution to the particular model. The adjustments (modifications) should be made in such a way that the intension of the pattern is preserved. During modifications to the original (and generic) solution, constraints imposed on the solution space in the forces section of the pattern template should not be violated. Also, the consequences of the particular solution after modifications should not be in contradiction to the consequences of the original pattern. One way of restricting the possible variations of the pattern solution is allow modifications that result of applying a set of operations that preserve the pattern properties [Pahl 2009a]. After performing such operations to an original pattern a validation step can be applied to check that properties have been preserved.

- *Pattern instantiation.* The instantiation of a pattern in a concrete model implies that certain elements and connectors in the model are designated to play the role of respective pattern elements (named pattern roles) and pattern connectors. In the case that certain pattern roles and connectors can not be represented by elements (or connectors) in the model, new model elements (or connectors) are created.

- *Pattern combination.* Several instances of a pattern or instances of different patterns might be present in a single process model. A combination of patterns can be used as a way to accentuate positive consequences of applying different patterns or to cover a broader problem that can be jointly addressed by a set of patterns. For each pattern, its effects (positives and negatives) on the rest of the applied patterns needs to be analysed. Since this is difficult to measure before implementation, at design-time, it is possible to analyse the potential interferences among documented consequences of each pattern description. If available, pattern languages define constrained ways to combine patterns that can result in non-conflicting solutions [Buschmann 2007]. More advanced approaches consider the use of formally defined operators to combine architecture abstraction. For example in [Pahl 2009a], the authors propose an ontology-based framework for style definition and style combination. Combination of styles is based on ontology relationships.

3.3.2 Service identification

This activity involves the analysis of process models and their relation to the software supporting the processes operation. The aim is to define the individual services that will compose the architecture solution. Among the different approaches to

define what services will be the building blocks of a service-based solution, two main categories can be identified. Approaches to identify *existing* services and approaches to design the scope and granularity of *new* services.

One approach to define the boundaries of a new process-centric service is one considering process patterns as the standard solutions to recurring operational problems. The process pattern identifies a set of common process elements and their relations in a particular process domain, it can identify common activities among different organisations – either within an enterprise or across different enterprises – and serves as guidelines to the definition of a reusable service across organisations [Gacitua-Decar 2009c].

The identification of new services based on process patterns consists of finding instances of one or more process patterns in a process model, and subsequently establishing new services based on each found pattern instance. The process model captures the operation of a business(es) and the new services would be designed to automate and/or integrate that business(es) operation. The composition of the new services would be governed by the process model. The pattern based techniques used to identify new services use as basis the identification of process patterns in concrete process models. Specifically, techniques for pattern matching and pattern discovery are proposed. They encourage the design of new services based on reusable abstractions at process level. While pattern matching takes a process model and a process pattern as input, pattern discovery only takes the process model. Both techniques provide as output a set of identified pattern instances in the input process models. The foundations and core solutions to implement the pattern matching and discovery techniques are detailed in Chapters 5 and 6.

3.3.3 Business model to service architecture transformation

This activity considers the incremental transformation of the augmented process models toward a process-centric service solution. During the transformation, the restrictions imposed by existing software – which supports the processes operation – are also considered. Pattern-based transformation templates are a medium defining a set of rules to transform predefined process model configurations into process-centric service compositions.

Note that several model-driven development approaches have followed a strategy of direct translation from business modelling constructs to software constructs, e.g., direct transformation from BPMN to BPEL constructs. However, business models could contain sections that cause deadlocks and other problems for the process execution [Koehler 2008b].

LABAS proposes the transformation from business models to service architec-

tures to be based on pattern-based transformation templates [Pahl 2009b]. An advantage of using transformation templates is that they can be designed to provide only error-free transformations. Nevertheless, this requires a previous step to refine the business process model to match with the business model section of the transformation template. In [Ouyang 2007], a control-flow pattern approach for BPMN to BPEL translation is presented. Transformation templates follow a similar approach, but are beyond control-flow structures.

3.3.4 Service architecture augmentation

This activity is similar to what was explained in Section 3.3.1, but instead process models, a service architecture model is augmented with SOA pattern instances. SOA patterns [Zdun 2006], [Erl 2008] provide guidelines to solve technical issues to design service-oriented architectures – such as services communication, security and distribution. There are approaches targeting automation during the incorporation of pattern instances into service architectures. For instance, in [Pahl 2006] a modelling and automated transformation approaches is proposed. It automatically generates executable web service compositions based on a distribution pattern chosen by a software architect and existing web service interfaces.

3.4 Traceability in LABAS

Traceability is an important aspect of the LABAS framework to maintain aligned modelling elements from different layers. A *traceability model* captures (in inter-layer and intra-layer models) the relationships between model elements and between pattern and model elements. These relationships cross different layers of abstraction and perspectives and they change over time. Based on instantiated patterns, the traceability model can be used to analyse the impact of changes on enterprise service-based architectures and to maintain aligned business and software levels. For example, assume the existence of models capturing a company's software infrastructure (and its location), applications, services and processes. Suppose now that the company has a cluster of servers in Moscow centrally running the main services supporting the company's accounting process. What are the implications of changing the centralised configuration style to a distributed one? How is the accounting application affected? What are the implications to the accounting process? Suppose that, the company's software services have been defined without a specific governance model and currently the company is planning to increase the amount of offered services. Software engineers in the company have been thinking of organising the services in different layers according to specific domains in an service repository. For that,

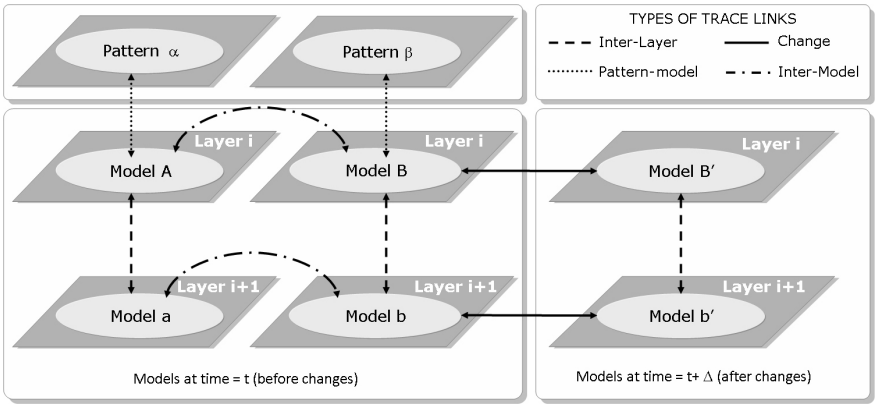


Figure 3.4: Types of trace links.

they plan to use the *service layers* and *domain inventory* patterns [Erl 2008]. What are the effects on other layers once those patterns are instantiated in the current service architecture? Also, after new services are created, are the benefits of the instantiated pattern maintained? The traceability model provides a medium to analyse the impact of changes on instantiated patterns and how these changes affect the service architecture solution and the processes it supports.

3.4.1 Types of Trace Links

There are different types of relationships between model and pattern elements in LABAS. The traceability model captures *dependency* relationships between model elements in different layers; *pattern instantiation* relationships from pattern to pattern instance elements; and the *evolution* of these dependency and instantiation relationships. Evolution of these relationships makes the traceability model dynamic and observation of this dynamic is core to the LABAS framework.

Dependency and instantiation relationships captured by the traceability model are modeled as *trace links*.

Trace links representing dependency relationships are further categorised in *inter-model trace links*, *change trace links* and *inter-level trace links*. Trace links between pattern elements and pattern instance elements are modeled as *pattern-model trace links*. Note that trace links between elements of executable models and elements of their instances at run-time are not captured in the LABAS's traceability model. Figure 3.4 illustrates the mentioned types of traceability.

- *Inter-model trace links* capture dependency relationships between elements in models from different perspectives but at the same level of abstraction. For example, a *send invoice* task in a business process model has a trace link to the

invoice element in a domain model. Both, the *send invoice* task and the *invoice* element are modeled at the same BML layer.

- *Inter-layer trace links* capture dependency relationships between model elements of the same perspective but at different layers of abstraction, for example elements from an abstract process model in BPMN [OMG 2008b] and its associated executable model in WS-BPEL [OASIS 2007]. Trace links between model elements from different perspectives and different layers of abstraction can be derived from inter-model and inter-layer traces.

- *Pattern-model trace links* capture the relationship between elements of a pattern and elements of a model. Such a model contains at least one pattern element instance. A *pattern-model trace link* is an special kind of *inter-layer trace link* since elements of the pattern and model are in the same perspective but pattern elements are at a higher level of abstraction. Certain modifications in model elements over time can trigger the firing of *trace rules* associated to *change trace links* whose consequences affect the modification or elimination of *pattern model trace links*.

- *Change trace links* capture changes of pattern instance elements over time. A single *change trace link* relates a pattern instance element in the same perspective and abstraction layer before and after a change. Successive *change trace links* capture a pattern instance element's change history. The definition of what corresponds to a change is defined by the model's maintainer. For instance, changes could correspond to predefined modifications in a "change" taxonomy as in [Briand 2006]. Specific conditions in *change trace links* can trigger the evaluation of *trace rules*. In turn, *trace rules* are associated to actions on pattern-model trace links that might unattach a pattern from one of its instances. More details on trace rules are given in Section 3.4.2.

3.4.2 Traceability Metamodel

A traceability metamodel defines the abstract syntax used to create the traceability model in LABAS. Figure 3.5 illustrates the main elements and relationships of the proposed metamodel. *TraceLink* and *TraceableElement* represent the connection and connected elements in the traceability model. A *TraceRule* can be evaluated when changes to a *TraceableElement* occur. According to results of the evaluation, changes can be propagated across layers of LABAS and keep aligned business and software levels. The proposed infrastructure allows the extension to new types of *TraceLink*s and specific *TraceRule*s. Moreover, there is no restriction to what kind of element is considered a *TraceableElement*.

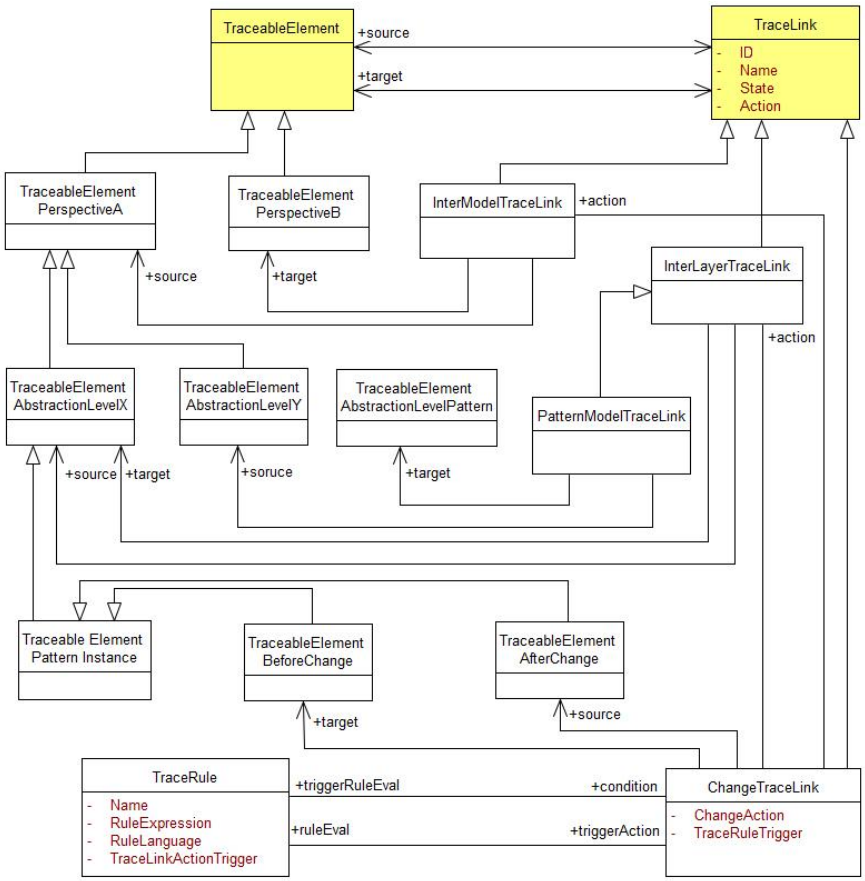


Figure 3.5: Traceability metamodel

- A *TraceLink* connects to *TraceableElement*s. A *TraceLink* has an *ID* and possibly a *Name* to identifying it. There are three types of *TraceLink*s according to what type of *TraceableElement*s are being connected. *TraceableElement*s from different perspectives but the same layer of abstraction can be connected through *InterModelTraceLink*. *TraceableElement*s from different layers of abstraction but the same perspective can be connected through *InterLayerTraceLink*, including *TraceableElement*s from abstractions such as patterns to be related to traceable model elements (pattern instance elements) by means of *PatternModelTraceLink*s. A *ChangeTraceLink* links two traceable elements of a modified pattern instance. One element is a pattern instance element in a recorded model before a change occurs and the other element is a copy or modified copy of that element in a next version of the model (after the change has occurred). Section 4.6.2 formalises and details on *pattern-instance changes*. A *TraceRuleTrigger* indicates if the evaluation of a *RuleExpression* in a *TraceRule* should be performed. For specific values of a *TraceLinkActionTrigger* in a *TraceRule*, a *ChangeAction* in a *ChangeTraceLink* could indicate that an action must be performed to set

the *State* of the *ChangeTraceLink*. Different values for *State* in a *ChangeTraceLink* could determine values for *ChangeAction* and eventually, eliminate the connection between a *TraceableElement* in a pattern model and a *TraceableElement* in a model.

- A *TraceableElement* defines a constituent part of a LABAS model, including elements in pattern configurations that can be connected through a *TraceLink* to other *TraceableElement*.

- A *TraceRule* represents a class of specialised rules used to evaluate the condition of pattern instances after a change. A *TraceRule* has a *RuleName* and defines a *RuleExpression* in a determined *RuleLanguage*. A *RuleExpression* is evaluated for determined values of *triggerRuleEval* which is associated to *condition* in *ChangeTraceLink*. After the evaluation of a *RuleExpression* in *ChangeTraceLink*, the *TraceLinkActionTrigger* can adopt different values and possibly trigger (through *ruleEval*) a *ChangeAction* in *ChangeTraceLink*. The triggered action could initiate the propagation of a change across *TraceableElement*s using *actions* associated to *InterModelTraceLink*s and *InterLayerTraceLink*s.

The previous sections have presented the different types of trace link in a metamodel that introduces an abstract syntax for creating a traceability model in the LABAS framework. The next section makes reference to how trace links are generated, stored and manipulated.

3.4.3 Trace Link Generation

There are different approaches to generate trace links between models, from manual efforts to the use of automatic techniques. While a common way of automatically generating inter-layer trace links involves instrumenting model transformations [Vanhooff 2007], [Aizenbud-Reshef 2006], there are also a number of approaches addressing manual generation of trace links across different layers. An automated trace generation mechanism for process-centric models could instrument a model transformation from business process models to executable processes. For instance, the (process) model transformation from models described in BPMN [OMG 2008b] to models in WS-BPEL [OASIS 2007] in [Ouyang 2007], could use an adaptation of the instrumentation strategy in [Vanhooff 2007]. However, as mentioned before in the related work chapter (Section 2.2), often business processes do not map one-to-one to services in a process-centric service architecture and ad-hoc adjustments need to be done to incorporate the constraints imposed by existing applications [Koehler 2008b], hence an approach for inter-layer traceability based on trace links

generated by end users can be more appropriate in this case. An example of providing modelling support for inter-layer traces is one followed by the Archimate framework [Lankhorst 2005]. The LABAS framework also use this type of approach to generate trace links between model elements, using modelling facilities provided by proposed UML-based profile. The profile is described in a subsequent chapter involving the evaluation and implementation of the framework (see Section 7.3.1).

For the particular case of inter-model trace links relating process model elements and domain model elements, they could be derived from specified pre- and post-conditions in domain model elements that are involved with process steps (process model elements). In [Jurack 2009], the authors use a graph transformation approach to define the semantics of refined activity diagrams with object flows. A control flow in a process is formalized by a set of transformation rule sequences and an object flow is described by partial dependencies between transformation rules. Since the informality of process and domain models in practice (available in organisations) could make approaches such as in [Jurack 2009] difficult to apply or ineffective, automated support for inter-model was left out of the scope in this work. While enterprise modelling begins to mature, plans for future work could incorporate the definition of a *formal* semantic for inter-model trace links.

Different strategies to generate and maintain trace links in several software development activities, including those making use of architectural abstractions such as patterns are discussed in [Lago 2009]. The authors indicated that there is a lack of approaches representing architectural abstractions explicitly and in an integrated way, and where and how these abstractions are used in concrete models. The LABAS framework attempt to overcome these needs by proposing a semi-automated method to create trace links between elements in concrete models and pattern configurations. This type of trace links are referred as pattern-model trace links. The method would allow end users to interact and select what trace links are generated. Initially, an end user would select a pattern(s) and a model(s). A matching technique would identify elements from the selected pattern(s) that are related to elements in the model(s). This step is performed by the same pattern matching technique involved with service identification, proposed in Section 3.3.2. Matched model elements can be highlighted and exposed to end users, which can choose what model elements will be finally traced to pattern elements. Trace links between model and pattern elements are maintained as part of the traceability model. This model can be used to manage changes across LABAS layers affecting the service architecture solution. This is illustrated with a case study in Section 7.2.2. The pattern matching technique used to create pattern-model trace links uses a graph-based algorithm [Gacitua-Decar 2009c] and semantic support [Gacitua-Decar 2009a]. The next chapters focus on the formal basis (Chapter 4) and detailed implementation of this technique (Chapter 5).

3.5 Summary

This chapter has explained the modelling context and organisation of the different layers of abstraction involved with service-based solutions for enterprise processes and applications integration. A pattern-based framework to address this integration problem was proposed. The framework consists of a layered architecture with business, services and application layers. Patterns and pattern-based techniques are central to the proposed approach for designing service-based architectures relating business processes and applications. A traceability model is used to maintain aligned model elements in different layers and to allow change impact analysis based on the effects of changing pattern instances in models.

Chapter 4

Graph-Based Process Models and Patterns

Contents

4.1 Process Models as Graphs

Process models often involve a graph-based representation used to facilitate communication and enhance understanding [Aguilar-Saven 2004]. Most graph-based representations consider graph vertices and edges as the basic elements to capture the structure and connectivity between process elements. A difference of other graph-based formalisms considering a graph as the state of a system and a process as a set of transformations from an initial to a final graph, such as in [Corradini 1996] or [Ehrig 1997], the graph-based representation of processes in this work considers a graph as the process itself. On edges, the flow of what is being processed and the order between two consecutive process steps (vertices) is directed. On vertices, changes to the system by processing or directing the flow and reacting to environmental conditions takes place. Structural specifications in graphs capture the execution order of

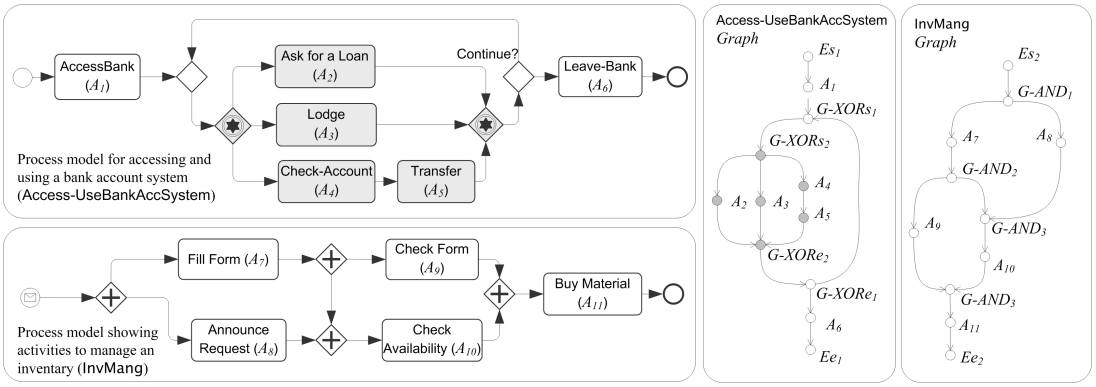


Figure 4.1: Two process models in BPMN v1.1 and related graph representations.

process steps and can evidence execution conflicts such as deadlocks and lack of synchronization [Sadiq 2000], but also abstractions such as patterns and their relation to process models can be represented through abstract types and constraints defined by a pattern graph. Structure preserving relations defined by a graph homomorphism between a process graph and a pattern graph can capture the relation between a pattern and its instances [Gacitua-Decar 2009c], similar to what a metamodel is to a concrete model [Ehrig 2008].

Figures 4.1 and 4.2 illustrate simple examples of graph-based representations for business process models and executable processes. Figure 4.1 shows two business process models annotated in BPMN [OMG 2008b], a well-known visual modelling notation for business processes. On the right-hand side, two graph representations are shown. Vertices and edges of the graph represent process model elements and their connections. The Use-AccessBankAccSystem process on the left has at its core (gray-colored elements) a common set of account usage activities that can be related to a process pattern. If so, the process pattern instance can be considered a subgraph of the Use-AccessBankAccSystem graph on the right (gray-colored vertices). Similar to business process models, executable process descriptions can also be associated to graph representations. Figure 4.2 illustrates an executable process description in WS-BPEL [OASIS 2007] (left-hand side) and a simplified graph representation (right-hand side).

The reminder of this chapter is dedicated to introduce a process model graph, a process pattern configuration graph, its instances, and to introduce the necessary graph-based foundations and notation. These special kinds of graphs are utilised in the next chapter to explain the proposed techniques for process pattern matching and discovery.

```
<bpel:process name="example" …">
<bpel:import … />
…
 +<bpel:partnerLinks>…</bpel:partnerLinks>
 +<bpel:variables>…</bpel:variables>
 -<bpel:scope>
  -<bpel:faultHandlers>
   -<bpel:catch faultMessageType="ns1:Exception" …>
    +<bpel:sequence>
     +<bpel:assign>…</bpel:assign>
      <bpel:reply name="Reply_S2" operation="…" …/>
    </bpel:sequence>
   </bpel:catch>
  </bpel:faultHandlers>
  -<bpel:flow>
   -<bpel:links>
     <bpel:link name="L1"/>
     <bpel:link name="L3"/>
     <bpel:link name="L2"/>
     <bpel:link name="L4"/>
   </bpel:links>
   -<bpel:receive … name="Receive_S1" …>
    -<bpel:sources>
       <bpel:source linkName="L1"/>
       <bpel:source linkName="L3"/>
     </bpel:sources>
   </bpel:receive>
   -<bpel:reply name="Reply_S3" …>
    -<bpel:targets>
       <bpel:target linkName="L2"/>
       <bpel:target linkName="L4"/>
     </bpel:targets>
   </bpel:reply>
   -<bpel:while>
    -<bpel:targets>
       <bpel:target linkName="L1"/>
     </bpel:targets>
     -<bpel:sources>
       <bpel:source linkName="L2"/>
     </bpel:sources>
     <bpel:condition>…</bpel:condition>
     +<bpel:sequence>
      -<bpel:assign>…</bpel:assign>
       <bpel:invoke … name="Invoke_I1" …/>
      -<bpel:assign>…</bpel:assign>
     </bpel:sequence>
   </bpel:while>
   -<bpel:sequence>
    -<bpel:targets>
       <bpel:target linkName="L3"/>
     </bpel:targets>
     -<bpel:sources>
       <bpel:source linkName="L4"/>
     </bpel:sources>
     -<bpel:assign>…</bpel:assign>
       <bpel:invoke … name="Invoke_I2" …/>
    -<bpel:assign>…</bpel:assign>
   </bpel:sequence>
  </bpel:flow>
 </bpel:scope>
</bpel:process>
```

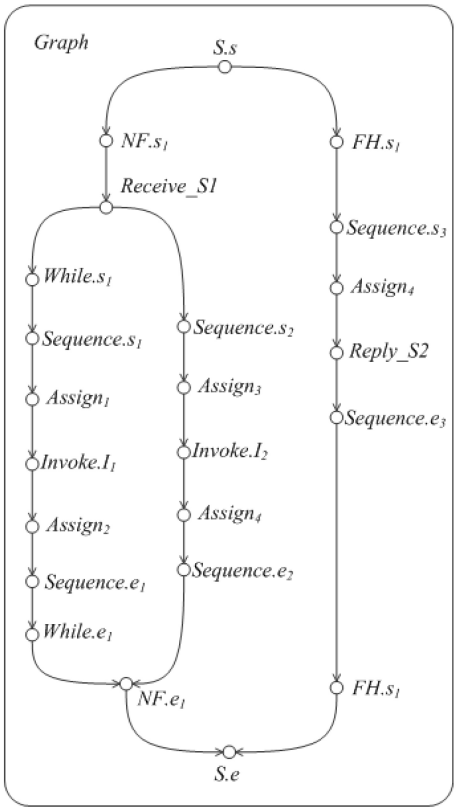


Figure 4.2: Excerpt of executable WS-BPEL process and related graph representation.

4.2 Process Model Graph

This and the next sections in this chapter are based on the background concepts and notation for graphs described in Appendix A. For more details please refer to that Appendix.

Let the graph representing a (business) process **model** be the typed attributed graph $M = \langle AM, am \rangle$ over ATM. The graph morphism $am : AM \to ATM$ relates the attributed typed graph AM to an attributed type graph $ATM = \langle TM, Z_M \rangle$ over Σ_M where Z_M is the final Σ_M-algebra Z_M. Vertex and edge types for M are represented by vertices and edges of TM, thus capturing types of process elements and their relations. Elements of Z_M represent the sorts of the signature which are included in TM as types for data vertices. Edges connecting abstract types to data types in ATM are attribute declarations.

Note that attributed type graphs can be enriched with inheritance relations and abstract vertices in a way such that a vertex type can inherit the attributes and edges of all its ancestors [de Lara 2007], [Taentzer 2005]. Also, constraints, such as cardinality on edges [Ehrig 2006b], can be added to ATM to define restrictions over any concrete model M.

An example of an attributed type graph (only an excerpt) associated to the specifications for creating process models in BPMN 2.0 [OMG 2009a] is illustrated in Figure 4.3. At the top of Figure 4.3, some of the elements defined in the specifications for BPMN 2.0 are shown. At the bottom of Figure 4.3 (right-hand side) a simplified example of a concrete process model with three activities in sequence ($A1$, $A2$ and $A3$) and its associated attributed typed graph M are shown. For the sake of simplicity, only graph and data vertices associated to types and attributes for the elements highlighted on the bottom/left-hand side of the figure are shown. The bottom/left-hand side of Figure 4.3 shows a section of the related attributed type graph ATM. *Activity* and *Sequence Flow* type vertices in ATM are derived from the BPMN 2.0 specifications (top of the figure), also attributes and data types. Note that vertex types have inherited the attributes and edges of their ancestors. For instance, vertices typed by the *Activity* and *SequenceFlow* vertices in ATM have inherited the attribute *name* (and associated data vertices *String*) from their common ancestor *FlowElement*. Note that in Figure 4.3 the types for elements and relations in the BPMN 2.0 specifications are considered graph vertices of ATM, simplifying the attribution for relation elements.

4.3 Process Pattern Configuration Graph

Similar to (business) process models, a pattern configuration of a process pattern description consists of a process model. The process model contains the pattern roles and their relations such that it provides a graphical representation of a generic solution to the operation problem addressed by the process pattern (Section 3.2.2). The pattern configuration can also identify constraints for values of pattern role attributes.

Let the graph representing a process **pattern** configuration be $P = \langle\langle AP, ap\rangle, Constr\rangle$ with $\langle AP, ap\rangle$ a typed attributed graph over ATP and $Constr$ a set of applicable constraints to $\langle AP, ap\rangle$. The graph morphism $ap : AP \rightarrow ATP$ relates the attributed typed graph AP to an attributed type graph $ATP = \langle TP, Z_P\rangle$ over Σ_P where Z_P is the final Σ_P-algebra Z_P. Vertices and edges of TP represent pattern role and pattern connector types. Attributes in TP (edges connecting abstract and data type vertices) are attribute declarations. Elements of Z_P represent the sorts of the signature which are included in TP as types for data vertices. There

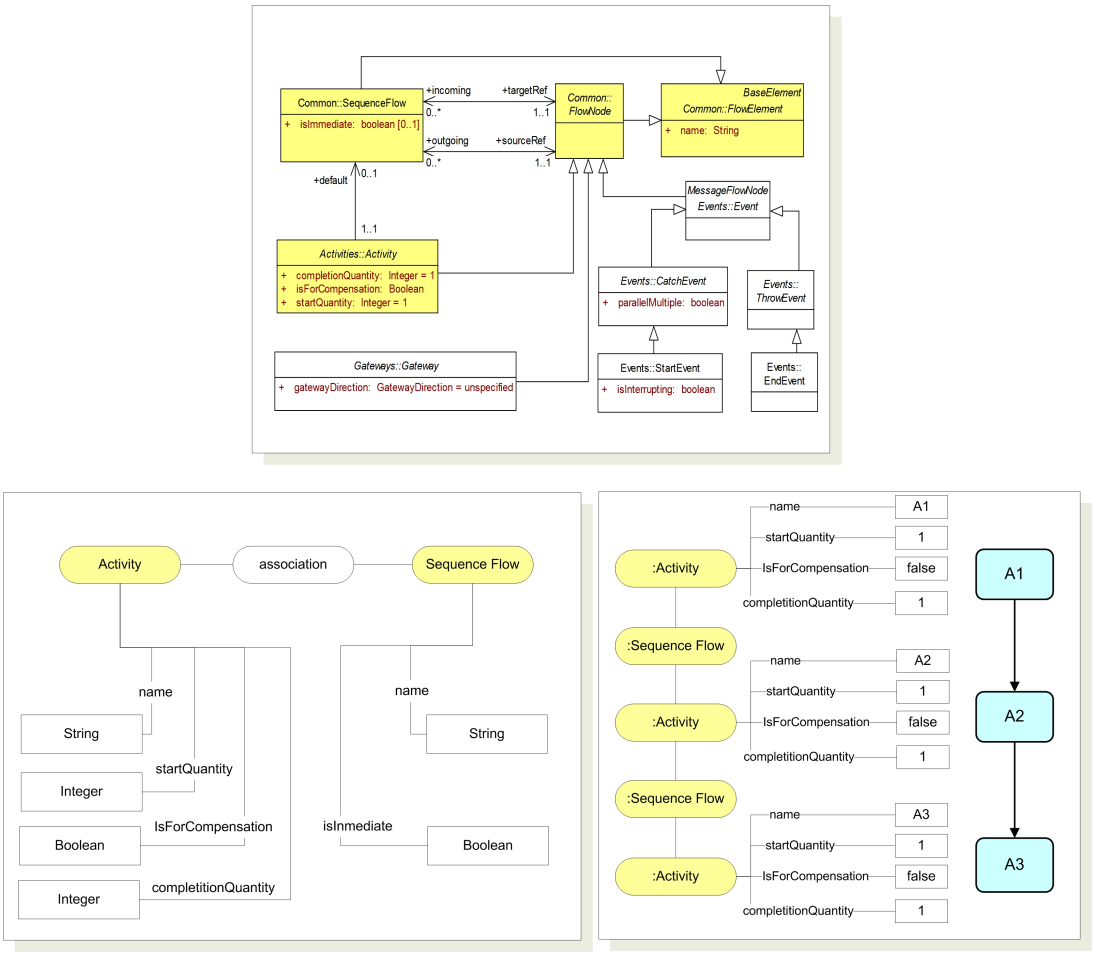


Figure 4.3: Excerpt of BPMN 2.0 specifications [OMG 2009a], associated attributed type graph and concrete process model example.

might be additional constraints for concrete data vertices such that attribute values are restricted to a subset of instances of a data type. Such constraints are captured in *Constr*.

Figure 4.4 illustrates a simple example of a pattern configuration P_y and its associated attributed type graph ATP_x defining abstract and data types. Concrete models that conform to the pattern must not only conform to the types of ATP_x but also respect the constraints for attributes values imposed by P_y. For instance, a *task* playing the role of *T1* should not last more than 30 minutes.

Note that the example in Figure 4.4 does not aim to illustrate any pattern used amongst practitioners, but provide an abstract example to explain the formalism above.

A more formal description of the relation between a process pattern (configuration) and its instances is addressed in the next subsection. The process pattern

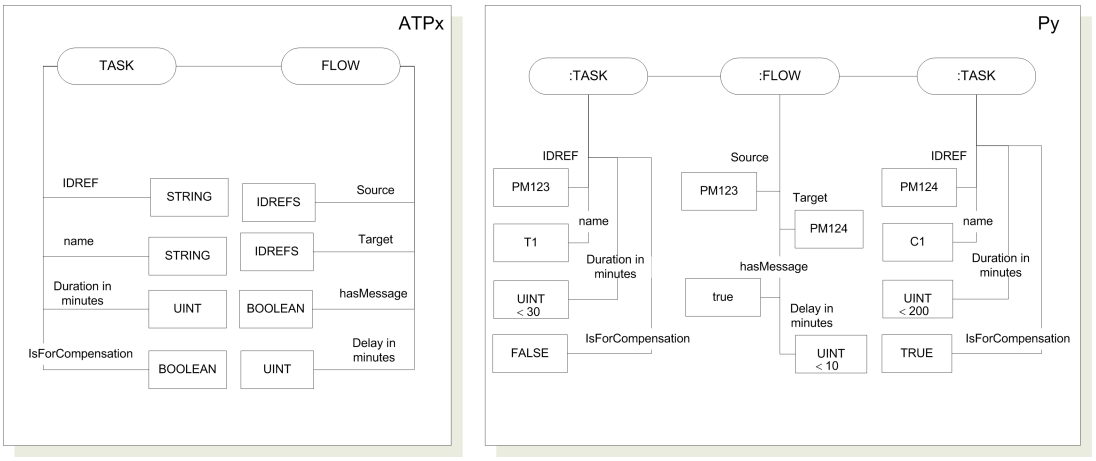


Figure 4.4: Example of pattern configuration and associated attributed type graph.

configuration is used by the proposed techniques in next chapters. In the remainder of the chapter, the term *process pattern* (or only *pattern* for short) refers to the graph-based representation of the process pattern configuration.

4.4 Process Pattern and its Instances

The concept and use of patterns in this work is related to the idea of providing a general solution to a frequent design problem. As described in more detail in Chapter 3, the description of a pattern contains (among other elements) a pattern configuration model. The pattern configuration model is an abstract model (architecture) representing a solution to the design problem addressed by the pattern. A *pattern instance* PI_i of a pattern configuration P in a particular model M is the *ith* concrete implementation of the generic (abstract) solution described by P over M. The abstraction relation between a pattern and its instances is similar to the relation between a metamodel and its concrete instances (models). The difference is in the addressed level of abstraction and scope. While a metamodel defines the abstract types, their structural constraints and data types for attributes in elements of concrete models; a pattern configuration defines the abstract roles, their structural constraints and restrictions of attribute values for elements of pattern instances present in a concrete model.

In a single model, several instances of a pattern can exist, as well as, several instances of different patterns. There is a special consideration when two or more instances of the same pattern or different patterns overlap in a model. This special condition is further discussed in Section 4.5.1. Now, the notion of *pattern instance graph* is introduced focussing on a single instance of a single pattern in a concrete model.

Suppose a process model M, as defined in the previous section (Section 4.2), and its associated ATM defining types and attributes for M. Also, suppose a process pattern configuration P as defined in Section 4.3. ATM defines the abstract types (and structural constraints) and data types for M; P defines the abstract roles and structural and data constraints for a subset of vertices and edges of M. The subset of vertices and edges conforming to types and constraints imposed by P defines the instance (or instances) of P in M. Intuitively, there are three levels of abstraction considered in this situation. One referring to a "metamodel" level, another associated to a "pattern" level, and (at the lower level of abstraction) there is a "model" level. Figure 4.6 illustrates this idea graphically. The two morphisms am and ap map vertices and edges from the metamodel[1] to the model level, and from the metamodel to the pattern level, respectively. These morphisms represent the structure preserving relations between the abstract and data types from M to ATM (am) and from P to ATP (ap). The morphism ai represents the preserving structural relations between a subset of vertices and edges in M to P. The subset defines a subgraph $API_i \subset M$ that, together with the morphism $ai : API_i \rightarrow P$, identifies the *ith pattern instance* of P in M.

4.5 Process Pattern Instance Graph

Let $PI_i = \langle API_i, ai \rangle$ over P with $API_i \subset M$ be an instance of the pattern configuration P in M, where $M = \langle AM, am \rangle$ over ATM and $P = \langle \langle AP, ap \rangle, Constr_P \rangle$ over ATP. The graph morphism $ai : API_i \rightarrow P$ relates the attributed typed graph $API_i \subset M$ to P, such that vertices and edges in API_i conform to the abstract types and structural and data constraints defined by the graph in P. Intuitively, elements and relations of the model section represented by API_i satisfy the constraints for pattern roles and connectors defined by the pattern configuration P. The relation between a vertex from P and vertices from API_i is not necessarily one-to-one. It can be one-to-many. This is explained in more detail in the next examples.

Consider the well-known design pattern *abstract factory* [Gamma 1995] for object-oriented programming (Figure 4.5). An abstract factory declares an interface for operations that create abstract product objects and concrete factories, who inherit from the abstract factory class, implement the operations to create concrete product objects. The number of concrete factories is not determined by the pattern, but it depends on its application. Thus, several concrete factories can be created in a model that instantiate the abstract factory pattern. Similar to the *abstract factory* pattern, the example in Figure 4.6 shows a single complete instance PI_i of P in M, which

[1] Note that the compact notation used in the specifications for BPMN 2.0, also illustrated in Figure 4.3 is used.

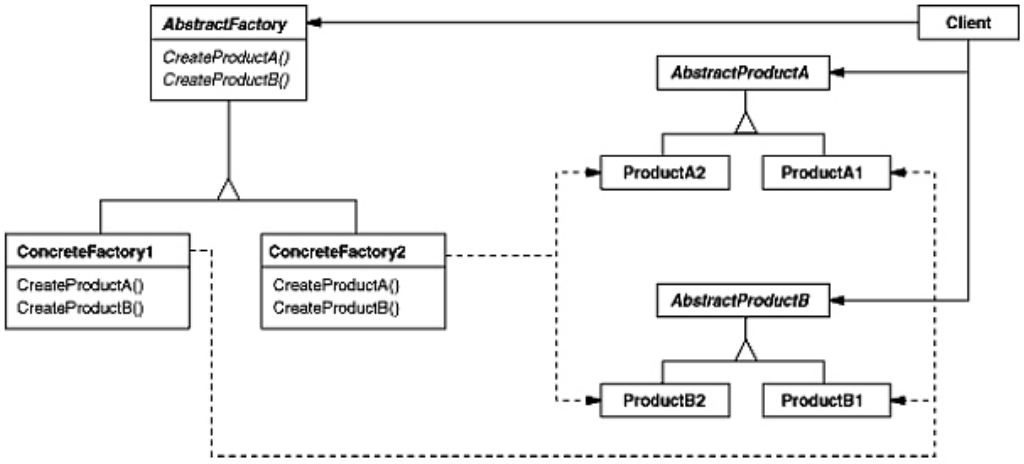


Figure 4.5: Configuration for abstract factory pattern [Gamma 1995].

has activities *Vote*, *Excuse* and *Invalidate* related to the pattern role *Action* from P through the mapping *ai*. This one-to-many relation captured by *ai* is fairly common for pattern instances. The morphism *ai* is said to be *surjective* and allows mapping several elements in PI_i to a single element in P.

The example[2] in Figure 4.6 shows a complete instance PI_i of P, however it is possible that some vertices and edges from M are mapped to a subset of all vertices and edges in P. Imagine that only the activity *Action* and gateways *Option* and *Continue?* are images of a mapping ai' relating elements from a pattern instance $PI_j \subset M$ to *some* elements from P. The morphism ai' is surjective but only maps partially to elements in P. Thus, $ai' : PI_j \to SP \subset P$ is called a partial (and surjective) morphism. The next chapter shall focus on a technique to automatically find complete and partial instances of patterns in models, including those related by surjective mappings.

4.5.1 Overlapping and Edge-disjoint Instances

The representation of process pattern instances as graphs would be utilised to define and implement techniques to identify pattern instances in models. One situation that can occur when identifying pattern instances is that instances can overlap. If they do not overlap, it may be of interest to identify new pattern instances in a model that is abstracted by reducing the non-overlapping instances to a special kind of vertices.

Let PI_1 and PI_2 be two instances of the pattern P_1 and P_2 in M, respectively. PI_1 and PI_2 are called *overlapped* if they have vertices in common, i.e. $V(PI_1) \cap V(PI_2) \neq \emptyset$. The graph $o(PI_1, PI_2)$ whose vertices are $V(PI_1) \cap V(PI_2)$ and edges

[2]Note that an enlarged figure of $ATM = ATP$ in Figure 4.6 is in Figure 4.3.

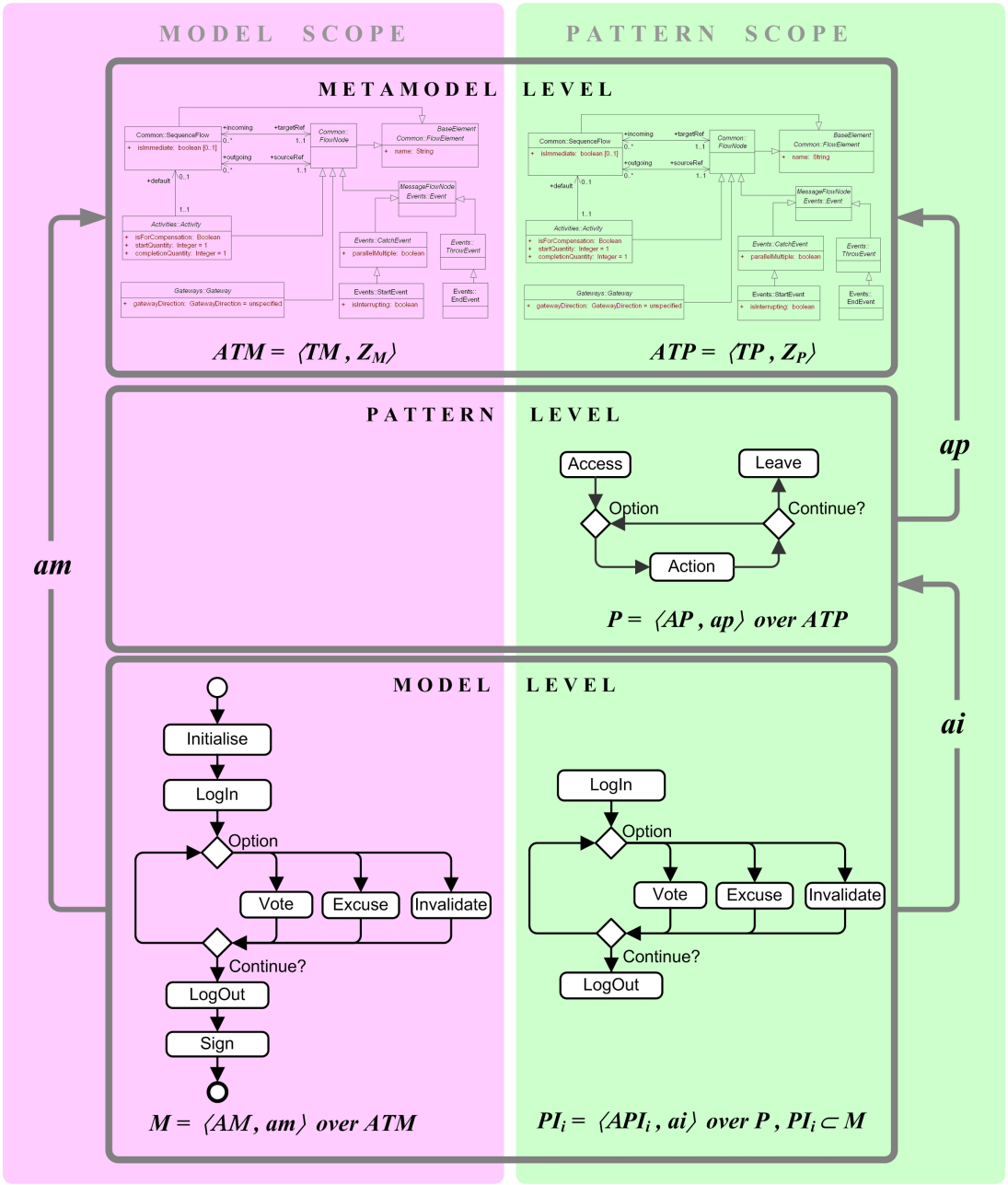


Figure 4.6: Model M, pattern configuration P and one of its instances P_i in M.

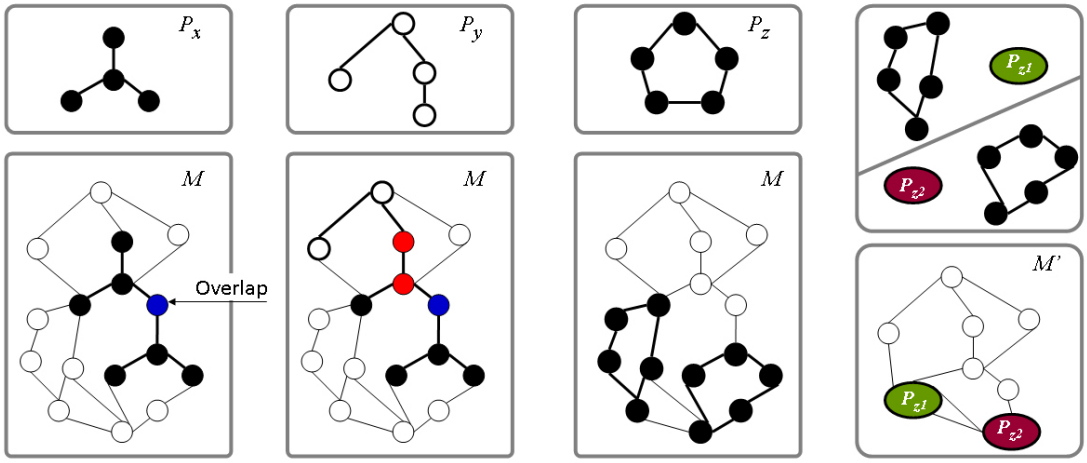


Figure 4.7: Illustration of overlapping and edge-disjoint instances.

$E(PI_1) \cap E(PI_2)$ is called the *overlap* between PI_1 and PI_2. PI_1 and PI_2 are called *edge-disjoint* if they do not share any edges of M, i.e. $E(PI_1) \cap E(PI_2) = \emptyset$.

If P_1 and P_2 are the same pattern, the instances PI_1 and PI_2 are called *identical* if for all edges $(u_i, u_j) \in E(PI_1)$, $(w_i, w_j) = (v_i, v_j)$, with $(w_i, w_j) \in E(PI_2)$ and for all vertices $u_i \in V(PI_1)$ and vertices $w_i \in V(PI_2)$, $v_i = w_i$.

Figure 4.7 (most left-hand side) illustrates an overlap in M between two instances PIx_1 and PIx_2 of a pattern Px. $o(PIx_1, PIx_2)$ is the single vertex indicated as an *overlap*. At the centre of the figure (left-hand side), an instance PIy_1 of the pattern Py is shown together with the previously highlighted PIx_1 and PIx_2. The overlaps $o(PIx_1, PIx_2)$ and $o(PIx_1, PIy_1)$ are also indicated. At the centre (right-hand side), two edge-disjoint instances PIz_1 and PIz_2 of Pz in M are shown. These two instances can be represented by two special vertices in M'. At the most right-hand side of Figure 4.7, the two special vertices are indicated with the same name assigned to the pattern instances, i.e. PIz_1 and PIz_2. M' is called the *raised* graph of M – or the *lifted* graph, as in [Fahmy 2000]. The two instances PIz_1 and PIz_2 together with the two edges connecting them are referred as a 2-fold cover of Pz.

Note that a formal definition of graph lifting is not detailed here, a number of articles regarding graph theory cover the topic at different levels. In [Fahmy 2000], a simplified explanation and practical use of graph lifting in the context of software architecture transformations is addressed, in [Diestel 2005] a more formal description of graph covering is provided. The basic concept of lifting a graph is used in the next chapter to explain a technique for hierarchical pattern matching (Section 5.4).

4.5.2 Model vs Pattern Attributed Type Graphs

An observation derived from Figure 4.6 regarding the attributed type graphs for M and P, i.e. ATM and ATP is discussed here. The example of Figure 4.6 shows that that M and P are both typed by the same attributed typed graph ($ATM = ATP$). However, this might not be always true in practice. For instance, if two different languages are used to describe the model and the pattern. The reason of using different modelling languages could be for example associated to preferences of different modellers or changes of language constructs over time.

In the next chapters, techniques to identify pattern instances in models are proposed. In their simplest form, these techniques assume $ATM = ATP$ or at least, the types mapped between M and P are not affected by the restrictions that P imposes on their instances in M. A more complex implementation of the techniques require an additional step to combine the attributed type graphs in a single graph defining types for pattern and model elements.

Suppose one or more instances of the pattern P exist in a model M. For a subgraph $PI_i \subset M$ to be an instance of P, there must exist not only the (surjective) morphism $ai : PI_i \rightarrow P$, but also the morphism $ap : PI_i \rightarrow ATP$ which defines the abstract and data types for P, given its associated ATP. Also, for M there exist a morphism $am : M \rightarrow ATM$, with ATM that defines the abstract and data types for M, and therefore for PI_i. On the other hand, a morphism between ATM and ATP does not need to be restricted to surjection nor injection, as long as the elements from both graphs (ATM and ATP) are not affected by the restrictions imposed by P. Figure 4.8 illustrates the latter. The figure shows two hypothetical alternative pairs of attributed type graphs for M and P – $\{ATM, ATP\}$ and $\{ATM', ATP'\}$. In the first case, for vertices in P and $PI_i \subset M$, the abstract type represented by a circle vertex in ATM is less restrictive than the abstract type represented by a circle-with-a-dot vertex in ATP, which adds an additional constraint (the dot). In the second case, in both ATM' and ATP', there is an abstract type represented by a circle vertex. While the concrete circle-with-a-dot vertex in PI_i is mapped to the circle-with-a-dot abstract type in ATM'; the concrete circle-with-a-dot vertex in P is mapped to the circle abstract type in ATP', however, it contains an additional restriction (the dot) at the pattern definition level.

4.6 Changes in Pattern Instances

The framework presented in the previous chapter contains a traceability model that maintains relations (trace-links) between model elements and elements in pattern configurations. Changes in models can affect the pattern instances and therefore

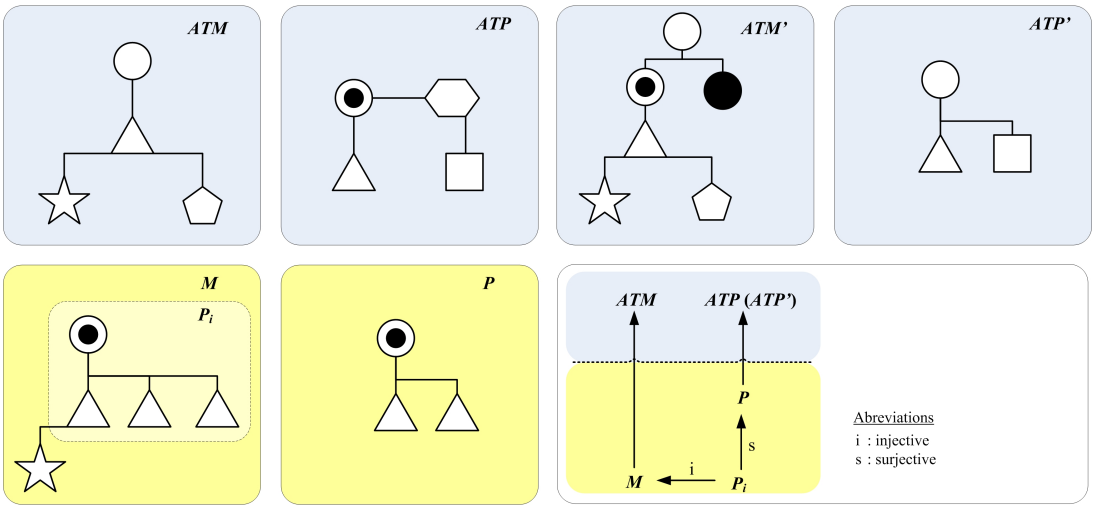


Figure 4.8: Model versus Pattern Attributed Type Graphs.

their relation to pattern configurations.

Consider a process model graph M and a process pattern graph P. Once P is instantiated in M, the host model M might be modified over time. These modifications could affect an instance PI_i of P in a way that it could be no longer an instance of P. There are two relevant elements regarding changes in pattern instances; they concern *what* is changed and *when* these changes occur. Awareness of time during modifications benefits the management of active executions of processes and provides an indication of the modifications' order. Identification of modified elements in a model and what type of modifications have been done provide important information to determine if pattern instances are being eroded – in a way that the benefits of the applied pattern could be lost. Moreover, if a determined ordered set of modifications is being performed (and they indicate that the pattern instance can be lost) actions to alert end users could be performed before the change that can erode the pattern instance to the point of losing it can be completed.

4.6.1 Recorded Models and Atomic Modifications

Atomic modification. Let *mod* be a (possible) model-level *atomic modification* of a model graph M. An atomic modification *mod* is a transformation of the graph M involving a single action (create, delete, modify) in either a vertex or edge of M. Note that (in general) M is an attributed typed graph and actions can be performed not only on graph vertices and edges, but also on attribute edges and data vertices.

Recorded Models and Change. Given a model M and a set of successive model modifications Mod_M, rec_M identifies a finite ordered set of *recorded graph models* de-

Figure 4.9: Relation between recorded changes and graph transformations.

noting successive stages (temporally ordered) from the original model M indicated by $M(t_0)$ to a final model indicated by $M(t_f)$. Each record $rec_M(t_i) \in rec_M$ stores the model graph $M(t_i)$ that refers to the state of the model M at instant t_i, i.e. $M(t_i)$ denotes the *ith* record of M after a set $Mod_i \subset Mod_M$ of successive modifications (change), such that the result of applying Mod_i to the previous graph in record $rec_M(t_{i-1})$ is equivalent to the transformation $Tmod_i : M(t_{i-1}) \rightarrow M(t_i)$. Thus, $Tmod_{t_i}$ identifies a *recorded change* from $M(t_{i-1})$ to $M(t_i)$.

$M(t_i)$ is an undirected graph where each of its vertices can store either the information from a vertex or an edge in M (M at instant t_i). A vertex in $M(t_i)$ that stores the information of an edge in M identifies its associated source and target vertices. Figure 4.9(a) illustrates the latter.

The instance t_i is a point in a *calendar* [Lanz 2009] C, where C is a set of absolute time points without gaps, such that C is a total order. The indexes for the family of transformations in $M(t_0) \Rightarrow^* M(t_f)$ is given by the ordered set of modification records rec_M such that each time a change is recorded, the set of modifications performed between a previous record $rec_M(t_{i-1})$ and a current record $rec_M(t_i)$ is identified in $M(t_0) \Rightarrow^* M(t_f)$ by the transformation $Tmod_i$. Figure 4.9(b) illustrates the latter.

4.6.2 Pattern-Instance Change

Let $PI_i = \langle API_i, ai \rangle$ over P with $API_i \subset M$ be an instance of the pattern configuration P in M, where $M = \langle AM, am \rangle$ over ATM and $P = \langle \langle AP, ap \rangle, Constr_P \rangle$ over ATP. P denotes the graph representing a process pattern configuration, $P = \langle \langle AP, ap \rangle, Constr_P \rangle$ over ATP and a set of applicable constraints $Constr_P$; and $M = \langle AM, am \rangle$ over ATM denotes the graph representing a (business) process model.

$M(t_0)$ and $PI_i(t_0)$ indicate the initial M and PI_i, respectively. An ordered set of records of M is denoted by rec_M, where $rec_M(t_j) \in rec_M$ contains the graphs $M(t_0)$ to $M(t_j)$ and refers to the derivation $M(t_0) \Rightarrow^* M(t_j)$.

A change in a pattern instance between two consecutive records of the model M is identified by the graph P, $PI_i(t_{j-1})$, $PI_i(t_j)$ (from two consecutive records $rec_M(t_{j-1}), rec_M(t_j) \in rec_M$) and by special edges representing pattern-model trace links and change trace links between P, $PI_i(t_{j-1})$ and $PI_i(t_j)$.

Pattern-model trace links and change trace links are part of the traceability model introduced in Section 3.4. The traceability model provides modelling support to maintain links (*pattern-model trace links*) between process pattern configurations and

its pattern instances in models, and to keep track of modifications in pattern instance elements through *change trace links*.

Change trace links connect elements before and after changes. In terms of the pattern instance graph PI_i of P in M and the model graphs $M(t_{j-1})$ and $M(t_j)$ from two consecutive records of M (i.e., $rec_M(t_{j-1})$ and $rec_M(t_j)$) a change trace link is an edge *ctl* connecting a vertex from $PI_i(t_{j-1})$ to its copy or modified copy in $PI_i(t_j)$ – but yet an instance of an element from P. Deleted vertices from $PI_i(t_{j-1})$ can not be linked. New created elements in $M(t_j)$, who are elements of the instance $PI_i(t_j)$, are traced to the corresponding pattern elements in P. Figure 4.10 shows change trace links and pattern-model trace links using the example graphs from Figure 4.8. From t_0 to t_1 the change involves eliminating an element (blue-coloured triangle) and associated relation, and from t_1 to t_2 the change involve eliminating an element (green-coloured triangle) and associated relation, and adding three new elements (white-coloured triangle, and two white stars) and their associated relations. Note that in order to simplify the figure, pattern-model and change trace links associated to edges in P were not depicted. Figure 4.10 at the bottom shows the pattern-model trace links and change trace links in the traceability model. A lifted view is also illustrated.

Pattern-instance change graph. Let M, P and rec_M be a model graph, a pattern configuration graph and records of M as introduced above. Let also PI_i be an instance of P in M. A *pattern-instance change graph* $PIC_i(t_j)$ is a graph capturing the history of recorded changes in PI_i, whose vertices $V(PIC_i(t_j)) = \{V(P) \cup \bigcup_{x=1}^{j} V(PI_i(t_x))\}$ and edges $E(PIC_i(t_i)) = \{E(P) \cup \bigcup_{x=1}^{j} E(PI_i(t_x)) \cup \{Ctl\} \cup \{PMtl\}\}$, where $\{Ctl\}$ is the set of all edges representing change trace links between $\{PI_i(t_0), \ldots, PI_i(t_j)\}$ and $\{PMtl\}$ is the set of all edges representing pattern-model trace links between P and $PI_i(t_0)$. Figure 4.10 at the bottom illustrates a pattern-instance change graph for P and one of its instances PI_i recorded at t_0, t_1 and t_2. A lifted view is also shown.

Differences between two consecutive graphs $M(t_{j-1})$ and $M(t_j)$ in rec_M are the result of atomic modifications over $M(t_{j-1})$. Edges in Ctl connecting vertices in $V(PI_i(t_{j-1})$ to vertices in $V(PI_i(t_j)$ have an attribute *State* (see traceability metamodel in Section 3.4.2). *State* can indicate if a vertex in $V(PI_i(t_j))$ has been modified with respect to a vertex in $V(PI_i(t_{j-1}))$. If a pattern instance element represented by v exists in $M(t_{j-1})$ and it is modified after t_j, there would be an edge connecting the vertex v in $V(PI_i(t_{j-1}))$ to its modified copy in $V(PI_i(t_j))$ (indicating the modification). If instead modifying v it is eliminated after t_j, there would not exist an edge from v in $V(PI_i(t_j))$ to a vertex in $V(PI_i(t_{j+1}))$. Also, if a pattern instance element

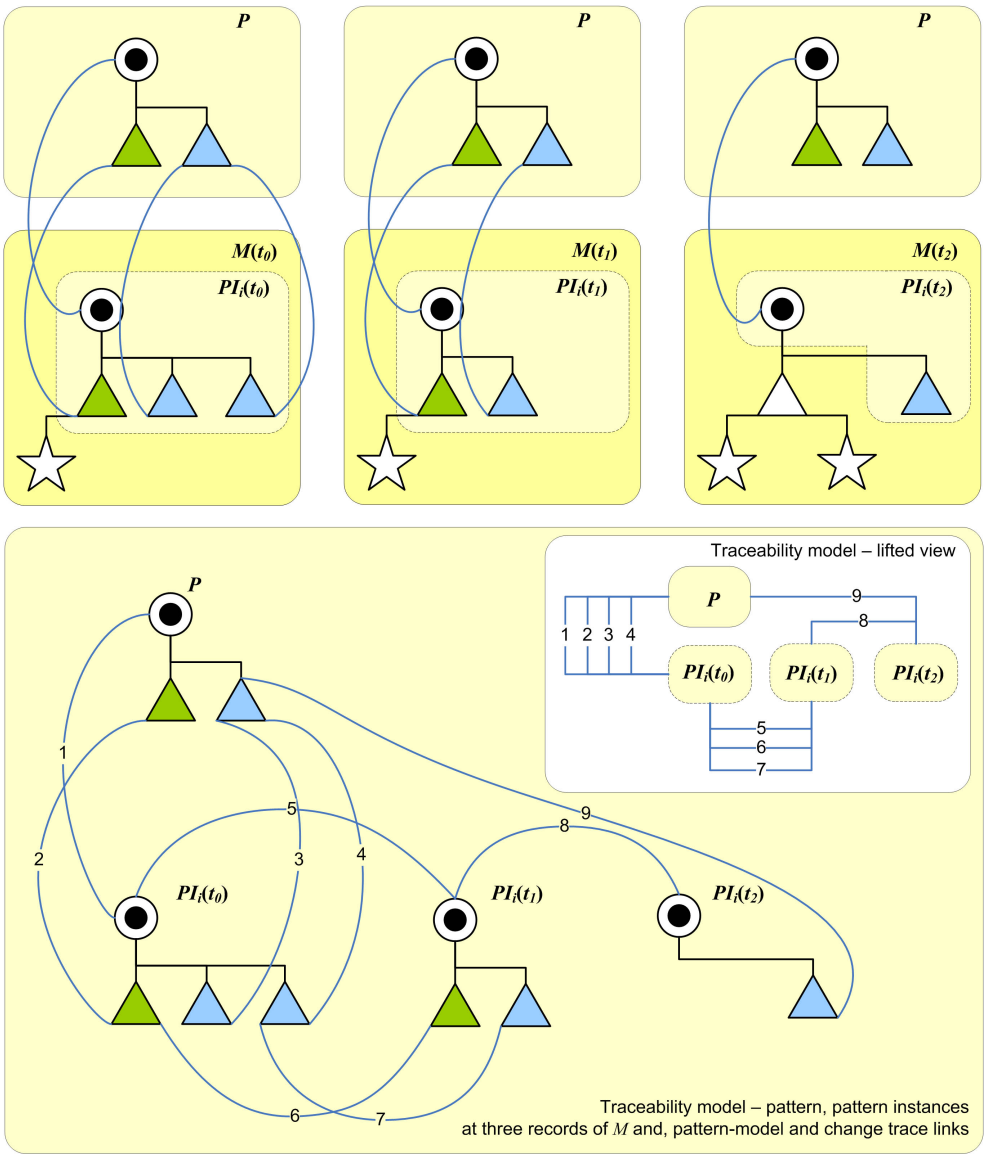


Figure 4.10: Pattern-instance changes and trace links in traceability model.

represented by w is created in M after t_j, an edge between w and its corresponding pattern role (or connector) in P is created and added to the set $PMtl$.

Explicit connections (edges) between P and its instances at different times can be used to identify modifications to pattern instances independently. This information can be used to support a pattern-based change impact analysis in a coarse granular way. In this case, granularity is given at pattern-instance level. The intention of working at this level and keeping independence between pattern instances is twofold. To use the information of pattern consequences to support change impact analysis and to have the possibility to analyse the system for possible independent changes on pattern instances.

In the context of the LABAS framework, the representation of pattern-instance changes as graphs can be used as a basis for a pattern modification technique used during augmentation of business process models and service architectures (Sections 3.3.1 and 3.3.4) and the use of the traceability model (Section 3.4) for change impact analysis. Chapter 7 demonstrates the use of the traceability model for a pattern-based change impact analysis. The aim is to facilitate the analysis of what properties of a system are affected with a change in a pattern instance. The assumption is that properties of systems can be related to patterns and therefore a pattern instance in a specific system's model would indicate that such property can be associated to the modelled system.

4.6.3 Conditions for Derived Pattern Instances

A pattern template that contains a pattern configuration P (as described in Section 3.2.3) could also add additional information regarding allowed and not allowed modifications for its instances. This can assist the modification of patterns as introduced in the pattern-based techniques of the LABAS framework.

Given an instance PI_i, a derivation from $PI_i(t_j)$ to $forbPI_i(t_{j+1})$ is a derivation that transform the pattern instance $PI_i(t_j)$ into a forbidden graph $forbPI_i(t_{j+1})$. The forbidden graph is a graph representing a pattern instance that interferes with the intentions of the pattern in such a manner that $forbPI_i(t_{j+1}) \subset M$ would no longer be an instance of P.

Consistency conditions, such as the existence or uniqueness of certain elements, are properties of graphs that have to be preserved by the application of rules (refer to Section A.6 in Appendix A for details in application and consistency conditions). Graphical consistency constraints as in [Heckel 1995] can describe conditions for existence or uniqueness of vertices and edges in graphs. The notion of consistency constraints and conditional applications over the left- and right-hand side of a graph transformation rule r (pre- and post-conditions over the rule r) can be used here

to define conditions on a rule transforming $M(t_j)$ to $M(t_{j+1})$, such that it does not induce the transformation $PI_i(t_j)$ to $forbPI_i(t_{j+1})$, which indicates a forbidden transformation from a valid pattern instance to an invalid one.

Conditions for a model derivation to hold a valid pattern instance. A direct derivation $M(t_{j-1}) \overset{r,m}{\Longrightarrow} M(t_j)$ can be restricted to the existence or non-existence of certain elements in $M(t_j)$. A restricted derivation can be defined for a rule $\hat{r} = (r, A(r))$ that defines pre- and post-conditions for the transformation rule r defined by $A(r) = (A_{L_{PI_i}}, A_{R_{PI_i}})$, where L_{PI_i} is a subgraph of $PI_i(t_{j-1})$ and R_{PI_i} is a subgraph of $PI_i(t_j)$. The rule $\hat{r}$ and a match $m : PI_i(t_{j-1}) \rightarrow M(t_{j-1})$ can define a conditional derivation $M(t_{j-1}) \overset{\hat{r},m}{\Longrightarrow} M(t_j)$ such that $M(t_j)$ contains a valid instance of PI_i. Other derivations that do not comply with $\hat{r}$ result in a model containing a forbidden subgraph $forbPI_i(t_{j+1})$.

Note that graphical consistency constraints can not express structural conditions like the existence of paths or cycles of arbitrary length or global graph properties such as connectivity. Also, edge multiplicity and inheritance, commonly used in visual languages, is not addressed in [Heckel 1995]. In [Taentzer 2005], consistency constraints are extended to graph constraints which allow the expression of multiplicity constraints and edge inheritance. Moreover, given a certain production, several matches might be possible and in that case, one of them has to be selected. There are several techniques to define the order to select determined matches to apply its associated production rules [Ehrig 2006a]. The aspects perviously mentioned are not within the scope of this work and can be considered as future work in the context of techniques for pattern modification and horizontal model transformations.

4.7 Summary

This chapter introduced graph-based representations for process graphs, process pattern graphs and process pattern instance graphs. It also defined the relation between pattern instances in process models and pattern configurations as graph morphisms. These representations are core to the proposed pattern matching and discovery techniques described in the next chapters. Special situations such as overlapping pattern instances and different attributed type graphs defining types for pattern and model elements were discussed. Changes in pattern instances were also captured in a graph-based representation. This representation supports the use of the traceability model defined in the context of the proposed LABAS framework to assist pattern-based change impact analysis. Future work regarding the pattern modification technique introduced in the previous chapter is also suggested. How-

ever, the emphasis of this work is on the graph-based pattern matching and discovery techniques.

Chapter 5

Pattern Matching

Contents

5.1 Overall Approach

Among the pattern-based techniques in the proposed framework for SOA design and integration (Chapter 3), pattern matching can be exploited to guide the definition of new services based on known and reusable process patterns.

The problem of matching a process pattern in a process model is addressed in this work as a *graph matching* problem. Process patterns and process models are represented by attributed typed graphs – introduced in Chapter 4. A process pattern

configuration graph and a process model graph are the input to the matching problem. The output is an annotated process model graph where pattern instances are identified.

This chapter explains a solution to the graph matching problem considering perspectives involving structure, hierarchy and semantics. The solution consists of a number of techniques, which are implemented as a family of algorithms. The algorithms implement structural matching by checking homomorphic relations between pattern and model graphs. Semantic matching considers the comparison between types and attributes associated to graph vertices from models and patterns. The algorithms go beyond exact matching and cover partial and inexact matching. This provides more flexibility for cases where models and patterns can originate from different sources.

The four main sections of this chapter cover a general description of the proposed techniques for pattern matching, their implementation as a family of algorithms, hierarchical matching of patterns and, finally, semantic matching at vertex level. Section 5.2 provides a formal description and indicates the relations between the different kinds of proposed structural matching approaches: exact, inexact, complete and partial. Formalisation can provide guaranties of correctness and improve confidence in tools. Section 5.3 explains the family of algorithms implementing structural matching. The algorithms include functions to match vertices from patterns and models by comparing their types and attributes. These functions can be enhanced by semantic similarity measures that are detailed in Section 5.5. After performing matching at one level of abstraction, other matching steps can take place at higher levels. A technique for recursively matching patterns at different levels of abstraction is proposed in Section 5.4. The technique is referred as hierarchical matching. Related work and concluding remarks are provided in the two final sections of this chapter.

5.2 Structural Matching

Graph matching has several years of investigation and solutions to this problem vary widely according to the specific graphs and type of matching required [Bunke 2005], [Conte 2004]. Solutions to the graph matching problem involve different complexities and implementations according to the kind of graph considered (directed, undirected, labelled, typed, attributed, static, dynamic, among others) and the type of matching required (e.g., exact, partial and inexact).

Graph matching involves the mapping between vertices of two graphs. From the graph structure perspective, this mapping must be edge-preserving; that is, for two vertices in one graph linked by an edge, they must be mapped to two vertices in

the other graph, also linked by an edge. A less restrictive form of matching does not need to ensure that vertices in one graph have to be mapped to distinct vertices in the other graph, allowing many-to-one correspondences. Even in this less restricted form of matching, the problem of graph matching is mostly NP-complete [Conte 2004]. Consequently, a solution to address the structural perspective of process pattern matching has to consider an efficient implementation.

In realistic scenarios, where processes and patterns could originate from a distributed and semantically heterogenous environment, *exact* pattern matching could be rather unlikely. Instead, *partial* and *inexact* matching acquires relevance. Figure 5.1 illustrates examples of *exact*, *partial* and *inexact* pattern instances of a process pattern. Figure 5.2 summarises the types of pattern matching. Exact, inexact and partial matching allow many-to-one correspondences. *Partial matching* identifies *exact* but *incomplete* pattern instances. Partial instances might exist due to modifications or evolution of previously instantiated patterns. Also, in cases where patterns have not been previously considered as part of a process design, partial matches might indicate an opportunity to improve the process by means of incorporating the whole design solution indicated by the pattern configuration. *Inexact matching* provides a *good*, but not *exact* solution to the matching problem. For instance, pattern instances can incorporate additional elements not described in the pattern, provided that they do not affect the documented consequences of applying the pattern (intention of the pattern). *Partial* and *inexact* pattern instances are relevant in practice, where process models and their implementations as services could be similar – although not exactly the same – from organisation to organisation. Identifying commonalities in process models, in this case by comparing pattern instances, can save costs and encourage reuse [Erl 2004].

5.2.1 Exact and Complete Process Pattern Matching

Matching is captured by the notion of graph homomorphism, which refers to a mapping having the property that, if two vertices form an edge in the source graph then their images form an edge in the target graph, where the target graph can also be the same source graph. Exact and complete matching is related to a weaker form of the most restricted form of matching, formally represented by an isomorphism.

Isomorphism indicates a bijective mapping between graph elements, i.e., a one-to-one correspondence must be found between each vertex from one graph to each vertex from the other graph – similar for edges. A weaker form of matching is *subgraph isomorphism*. It requires that an isomorphism holds between a source graph and a vertex-induced subgraph of the target graph. Yet another form of subgraph matching is *surjective subgraph homomorphism*, where surjection applies to both vertex- and

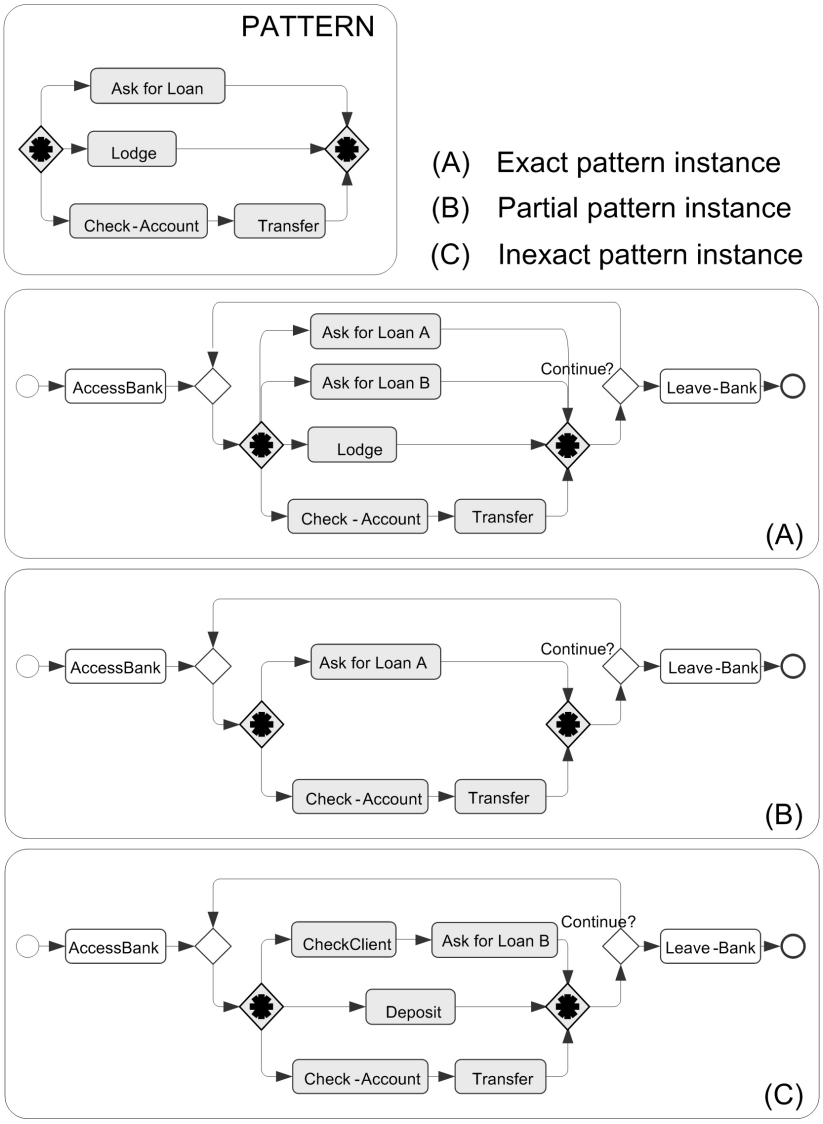


Figure 5.1: Exact, partial and inexact pattern instances.

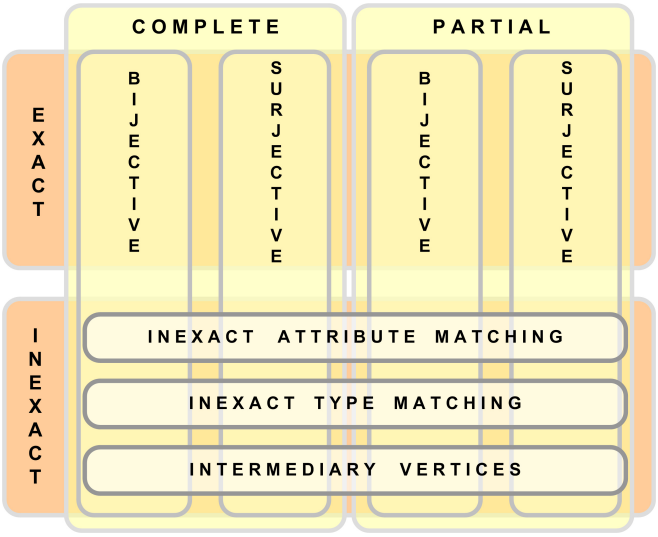


Figure 5.2: Types of pattern matching.

edge- mappings between source and target graphs. A special kind of this type of homomorphism are the *locally constrained graph homomorphisms* (LCGH), where the image of the neighbourhood of a (source) vertex in a graph is contained in the image of source vertex's neighbourhood in the target graph [Fiala 2008]. Exact and complete matching is formalised by this last type of graph homomorphism. For more details, refer to Annex A describing the mentioned types of graph homomorphisms.

Now, let M and P be a process model graph and a process pattern graph, respectively. M is a typed attributed graph $M = \langle AM, am \rangle$ over ATM and P is a typed attributed graph $P = \langle \langle AP, ap \rangle, Constr \rangle$ over ATP and a set of applicable constraints $Constr$. Consider that M_t and P_t are the recorded graphs. M_t and P_t are undirected graphs. Original edges in these graphs have been replaced by vertices with the same attribute, type and source/target information (see Section 4.6.1 for details).

An *exact process pattern match* between an instance PI_j of P in M is captured by a mapping $ecm : PI_j \rightarrow P_t$, where $PI_j \subseteq M_t$ is the jth instance of P in M. The mapping ecm stands for *exact complete mapping* and refers to a locally constrained homomorphism between PI_j and P_t.

As already mentioned, process pattern instantiation can also involve many-to-one relations between several elements from a model and a single element from the pattern. This situation is captured by allowing surjection in the graph homomorphism. Surjection (or bijection) is indicated by using $ecsm$ (or $ecbm$) instead ecm, being the exact complete mapping *surjective* or *bijective* mapping, respectively.

5.2.2 Exact and Partial Process Pattern Matching

Non-complete (partial) instances of patterns can exist in a model, for example, in cases where sections of pattern configurations have been applied unintentionally. Figure 5.3 illustrates an example with two process patterns - one abstract process pattern *send-receive message* (*P1*) and a more domain specific pattern for a *transfer transaction* (*P2*) - and two complete instances of *P1* and a partial instance of *P2* in a business process model. The pattern role *Check Clients Funds* from *P2* has no instances in the model.

Note that patterns not only occur as business process-level patterns. They can have refined versions at (lower abstraction) service implementation level or simply occur at this level. For instance, *P*1 can be refined to the *Request-Reply* pattern implemented in a JMS (Java Message Service) in [Hohpe 2004]. To apply the pattern matching technique at lower level, the service description would be transformed into a graph-based description. Similar to how a business process description is transformed to a graph-based representation.

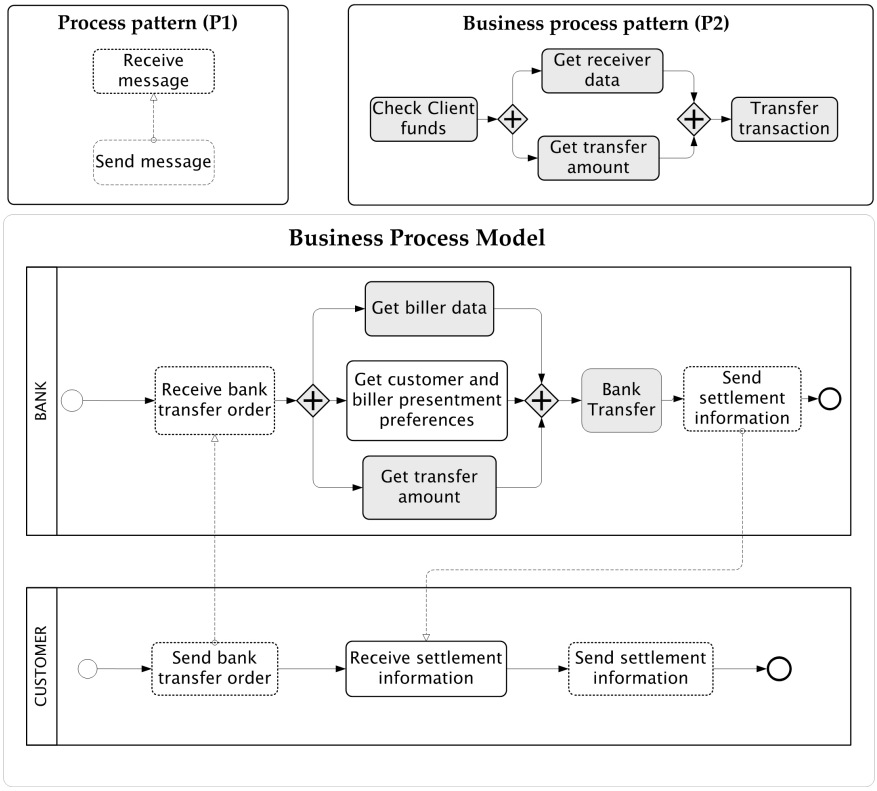


Figure 5.3: Example of partial process pattern instance.

For M and P as above (Section 5.2.1), a *partial process graph match* refers to a mapping from a subgraph SPI_j in M_t to a subgraph SP in P_t. SPI_j is called the jth

partial instance of P and it can be seen as a subgraph of an hypothetical complete instance PI_j in M_t. The mapping epm maps elements from SPI_j to a reduced number of elements in the codomain of $ecm : PI_j \rightarrow P_t$. As ecm, the exact partial mapping $epm : SPI_j \rightarrow SP$ is also type preserving. The difference is in the restricted domain and codomain of the mapping. Surjective and bijective mappings are indicated by $epsm$ and $epbm$, respectively.

5.2.3 Inexact and Complete Process Pattern Matching

In some – often frequent – circumstances, the constraints imposed by an *exact* matching are overly inflexible for the identification of model sections that are somehow *similar* to exact pattern instances. In several scenarios, the observed graphs are subject to deformations due to causes such as intrinsic variability of instances (implementations) of pattern configurations; errors during the acquisition/documentation of processes; presence of nondeterministic elements when translating processes to graph representations (e.g., unclear assignment of activities to roles in choreographies, unknown service support for composed services) among possible causes for having actual graphs that differ from their ideal models. Process pattern matching has to be tolerant to inaccuracies in observed graphs, by relaxing to some extent the constraints defined by an exact matching [Conte 2004].

Let M and P be the model and pattern graphs of the two previous subsections, and let PI_j and $ixPI_j$ be a complete exact instance and a complete *inexact instance* of P in M. An *inexact complete match* refers to a pair of mappings $\langle ecm, sim \rangle$, with $ecm : PI_j \rightarrow P$ as in Section 5.2.1, and $sim : \{x, y\} \rightarrow (t, a)$, where the mapping sim relates a pair of vertices $\{x, y\}$ with $x \in V(ixPI_j)$ and $y \in V(PI_j)$ to a pair (t, a), with $t, a \in \mathbb{R}_0^+$.

The pair (t, a) indicates a measure of *similarity* between types (t) and associated attributes (a) of x and y. It is expected that values of (t, a) are within a certain threshold defined in the description of elements of P. *Similarity* between graph vertices has different connotations according to what is measured [Blondel 2004], [Leicht 2006]. To introduce vertex similarity, additional background information and formalisation is required. Section 5.5 explains vertex similarity for pattern matching. This section continues to focus on the structural relationship between elements of patterns and their (inexact) pattern instances.

Intermediate elements. Consider the pair $\langle ecm, sim \rangle$ indicating an inexact and complete process pattern match of P in M. $ixwiPI_j$ refers to an inexact and complete process pattern instance *with intermediate* elements allowed. Figure 5.1(C) shows a simple example of an inexact pattern instance with a single intermediate element,

i.e., the CheckClient activity. Intermediate elements relax the mapping *ecm* (to a mapping *ecm'*) such that other elements not mapped in *ecm* and located in one or more edges connecting elements in the original domain of *ecm* are allowed as part of the pattern instance $ixwiPI_j$. An *intermediate* element is considered to be a subgraph of $ixwiPI_j$ that is not mapped in *ecm*. Intermediate elements should not interfere with the pattern intention and they could be identified from descriptions of the pattern configurations, where allowed modifications to patterns could be described in the associated pattern templates.

5.2.4 Inexact and Partial Process Pattern Matching

Similar to Section 5.2.2, non-complete (partial) and inexact instances of patterns could appear in a model after its modification or in cases where sections of pattern configurations have been applied unintentionally.

An *inexact and partial process pattern match* refers to a pair of mappings $\langle epm, sim \rangle$, where the domain of the original *sim*, indicated in the previous section, have been restricted. Here, the mapping $sim : \{x, y\} \rightarrow (t, a)$ considers an $x \in V(ixSPI_j)$ such that $V(ixSPI_j) \subseteq V(ixPI_j)$ and $y \in V(SPI_j)$ with $V(SPI_j) \subseteq V(PI_j)$. $ixSPI_j$ is called the jth inexact and partial instance of P, where SPI_j is the associated - hypothetical - exact and partial instance. Vertices in $V(ixSPI_j)$ are related to vertices in $V(SPI_j)$ by its degree of similarity indicated by *sim*. Similar to $ixPI_j$, *intermediate* elements can be added to edges connecting two elements in $ixSPI_j$. The *inexact and partial pattern instance with intermediate elements* is denoted by $ixwiSPI_j$, and is related to the pair of mappings $\langle epm', sim \rangle$, where $epm' : ixwiSPI_j \rightarrow SP \subseteq P$ and *sim* as above.

5.2.5 A Comprehensive Pattern Matching Framework

A comprehensive framework for pattern matching should consider not only exact and complete instances of patterns in models, but also possible partial and inexact instances. Empirical studies have shown that variants of process patterns – not exactly matching a pattern definition – can exist in real process models [Thom 2009], [Smirnov 2009]. The same situation has been considered in for pattern matching in other domains that use a graph-based approach [Gallagher 2006b], [Conte 2004].

Figure 5.4 illustrates an abstract pattern (left) and five types of pattern instances (right) that represent the types/attributes of matches indicated in the four previous sections. Different shapes identify different vertex types/attributes. Identical shapes with different fillings identify similar (but not equal) types/attributes. This abstract example is used to show in a compact figure the different notations for mappings involved in complete, partial, exact and inexact pattern matches. *ecm* indicates exact and complete matching, *epm* exact and partial matching, $\langle ecm, sim \rangle$ inexact and

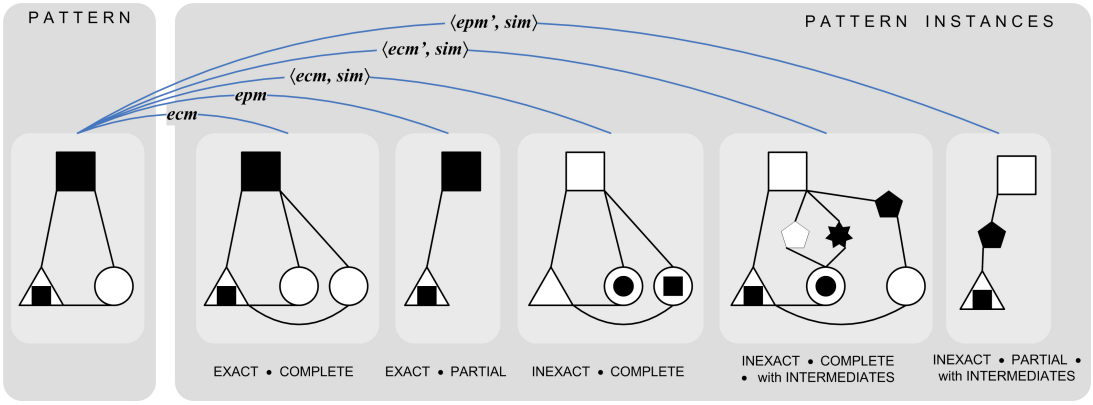


Figure 5.4: Kinds of pattern instances (complete/partial, exact/inexact, with intermediate elements).

complete matching, $\langle ecm', sim \rangle$ inexact and complete with intermediates matching and $\langle epm', sim \rangle$ inexact and partial with intermediates matching.

The next section explains the implementation of a pattern matching technique that considers different types of matching and uses a family of pattern matching algorithms.

5.3 Algorithms for Structural Matching

Let M_t and P_t be two graphs representing the recorded graphs of the process model and process pattern graphs M and P, respectively. Two algorithms are proposed for exact and inexact matching, allowing complete and partial matches of P_t in M_t:

- The CP-E-PM algorithm is used to identify complete, partial and exact instances.
- The CP-I-PM algorithm is used to identify complete, partial and inexact instances.
- The H-PM algorithm is used to recursively find instances of patterns at different abstraction layers, after lifting the pattern instances host model in each iteration.

Prior to the introduction of the algorithms, matrix-based structures to manipulate process and pattern graphs are described. These matrix-based structures facilitate the implementation of the algorithms in an environment with improved performance to work with matrices and basic functionality to manipulate them. This separates what specifically distinguishes the (pattern) graph matching algorithms from basic functionality such as accessing/modifying vertices information or deleting/creating

vertices. Section 8 and Appendix D refers to tool implementation and evaluation details.

5.3.1 Matrix-based Structure for Process and Pattern Graphs

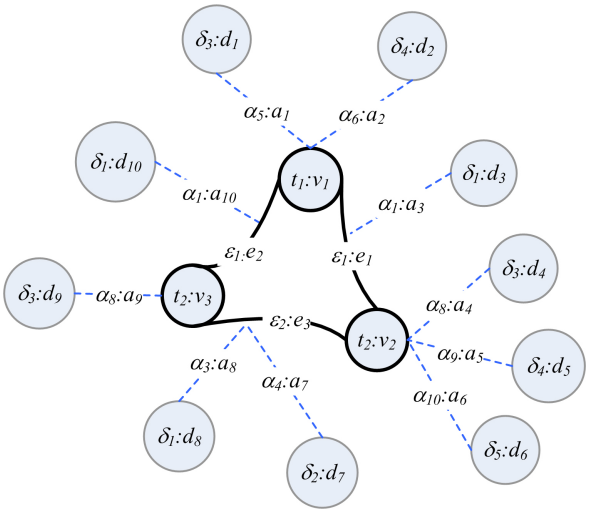
A number of matrix-based structures are involved in the implementation of the pattern matching algorithms. These are an *adjacency matrix* with rows and columns indicating the connectivity between pattern and model graph vertices, an *attribute matrix* whose rows are indexed by attributes and columns identifying graph vertices, and two matrices *abstract type matrix* and *data type matrix* whose rows are indexed by abstract and data types and columns indicating graph and data vertices, respectively. Note that adjacency-, attribute- and abstract/data type- matrices can be implemented as sparse matrices to save memory space, but here the full structure is referred to for simplifying the explanations.

Before introducing the matrix structures, the sets of graph vertices and edges from models and patterns are categorised. Let G_t represent a recorded graph of a process graph G, with G a typed attributed graph $G = \langle AG, ag \rangle$ over ATG. ATG is the attributed type graph defining abstract types of G vertices and data types for attributes of G vertices. Note that, as explained in Section 4.6.1, edges of G are stored as vertices in G_t. The sets of vertices and edges of G_t and ATG are subdivided as follows.

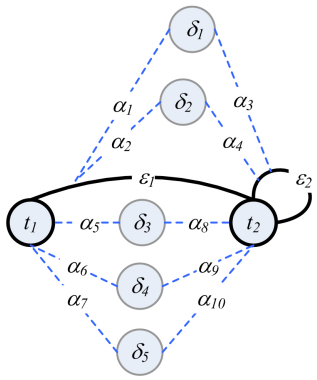
- $gV(G_t)$ denote graph vertices in $V(G_t)$,
- $aV(G_t)$ denote attribute vertices in $V(G_t)$,
- $dV(G_t)$ denote data vertices in $V(G_t)$,
- $gV(ATG)$ denote abstract types in $V(ATG)$,
- $dV(ATG)$ denote data types in $V(ATG)$,
- $gE(ATG)$ denote graph edges in ATG,
- $aE(ATG)$ denote attributes in ATG.

Note that $V(G_t) = \{gV(G_t) \cup aV(G_t) \cup dV(G_t)\}$ and $V(ATG) = \{gV(ATG) \cup dV(ATG)\}$.

Examples of an adjacency matrix ($Adj(G_t)$), attribute matrix ($Attr(G_t)$), abstract type matrix ($AType(G_t)$) and data type matrix ($DType(G_t)$) of a recorded graph G_t are shown in Figure 5.5. The graph G, its associated attributed type graph ATG and the matrices are indicated in the figure. The different matrix structures are explained in the next paragraphs.



(a) Graph G



(b) Graph ATG

	v_1	v_2	v_3	e_1	e_2	e_3
v_1	0	0	0	1	1	0
v_2	0	0	0	1	0	1
v_3	0	0	0	0	1	1
e_1	1	1	0	0	0	0
e_2	1	0	1	0	0	0
e_3	0	1	1	0	0	0

(c) $Adj(G_t)$

	α_1	α_3	α_4	α_5	α_6	α_8	α_9	α_{10}
v_1	0	0	0	d_1	d_2	0	0	0
v_2	0	0	0	0	0	d_4	d_5	d_6
v_3	0	0	0	0	0	d_9	0	0
e_1	d_3	0	0	0	0	0	0	0
e_2	d_{10}	0	0	0	0	0	0	0
e_3	0	d_8	d_7	0	0	0	0	0

(d) $Attr(G_t)$

	t_1	t_2	ε_1	ε_2
v_1	1	0	0	0
v_2	0	1	0	0
v_3	0	1	0	0
e_1	0	0	1	0
e_2	0	0	1	0
e_3	0	0	0	1

(e) $AType(G_t)$

	δ_1	δ_2	δ_3	δ_4	δ_5
α_1	1	0	0	0	0
α_2	0	1	0	0	0
α_3	1	0	0	0	0
α_4	0	1	0	0	0
α_5	0	0	1	0	0
α_6	0	0	0	1	0
α_7	0	0	0	0	1
α_8	0	0	1	0	0
α_9	0	0	0	1	0
α_{10}	0	0	0	0	1

(f) $DType(G_t)$

Figure 5.5: Sample graph G, associated ATG and $Adj(G_t)$, $Attr(G_t)$, $AType(G_t)$, $DType(G_t)$ matrices.

Adjacency Matrix. The adjacency matrix of G_t is denoted by $Adj(G_t)$ and has dimension $|gV(G_t)| \times |gV(G_t)|$. $Adj(G_t)$ is symmetric and each of its non-diagonal entries $Adj(G_t)_{ij}$ indicates the existence or non existence of an edge from vertex i to vertex j, with both vertices in $V(G_t)$. Diagonal entries having a value equal to 0 indicate that the associated vertices do not have self-loops in G_t.

$$Adj(G_t)_{ij} = \begin{cases} 1 & \text{if } i,j \in E(G) \text{ and } i,j \in V(G) \\ 0 & \text{otherwise} \end{cases}$$

Abstract Type Matrix. The abstract type matrix of G_t is denoted by $AType(G_t)$ and has dimension $|gV(G_t)| \times |gV(ATG)|$. $AType(G_t)$ is generally asymmetric and each of its entries $AType(G_t)_{ij}$ indicates that the graph vertex i is of type j.

$$AType(G_t)_{ij} = \begin{cases} 1 & \text{if } ag(i) = j \text{ for all } i \in gV(G_t) \text{ and } j \in gV(ATG) \\ 0 & \text{otherwise} \end{cases}$$

Attribute Matrix. The attribute matrix of G_t is denoted by $Attr(G_t)$ and has maximum dimension $|gV(G_t)| \times |aE(ATG)|$, with columns varying according to existing attributes for vertices in $gV(G_t)$. $Attr(G_t)$ is generally asymmetric and each of its entries $Attr(G_t)_{ij}$ indicate the attribute value x for attribute i of a graph vertex j. Note that data values (entries in $Attr(G_t)$) have various types. Those types are indicated in the data type matrix ($Dtype(G_t)$).

$$Attr(G_t)_{ij} = \begin{cases} x & \text{if } ji, ix \in E(G_t),\ j \in gV(G_t), \\ & i \in aV(G_t) \text{ such that } ag(i) = kl, \\ & \text{with } kl \in aE(ATG), k \in gV(ATG), l \in dV(ATG) \text{ and} \\ & x \in dV(G_t) \text{ such that } ag(x) = y, \text{ with } y \in dV(ATG) \\ 0 & \text{otherwise} \end{cases}$$

Data Type Matrix. The data type matrix of G_t is denoted by $DType(G_t)$ and has dimension $|aE(ATG)| \times |dV(ATG)|$. $DType(G_t)$ is generally asymmetric and each of its entries $DType(G_t)_{ij}$ indicate that data values for attributes in $aV(G_t)$ whose (attribute) type is i are of (data) type j.

$$DType(G_t)_{ij} = \begin{cases} 1 & \text{for all } i,j \in aE(ATG),\ j \in dV(ATG) \text{ and} \\ & i \in gE(ATG) \vee gV(ATG) \\ 0 & \text{otherwise} \end{cases}$$

5.3.2 Complete/Partial & Exact Pattern Matching Algorithm

Let M_t and P_t be the recorded graphs of the process model graph $M = \langle AM, am \rangle$ over ATM and the pattern graph $P = \langle \langle AP, ap \rangle, Constr_P \rangle$ over ATP. The adjacency-, attribute- and abstract/data type- matrices for M_t and P_t are denoted by $Adj(M_t)$, $Adj(P_t)$, $Attr(M_t)$, $Attr(P_t)$, $AType(M_t)$, $AType(P_t)$, $DType(M_t)$ and $DType(P_t)$, respectively. Note that $Attr(P_t)$ includes constraints for each of its entries according to $Constr_P$, if any.

Note that trace links between identified pattern instances of P_t in M_t and the P_t can be generated based on the pattern matching algorithm. Once trace links are established after a match, changes in the model could be tracked to check if pattern instances have been modified and the intention of the pattern associated to P_t has been violated (see Section 3.4 for more details).

CP-E-PM consists of two main stages:
A first stage identifies individual instances of vertices in $gV(M_t)$ such that they preserve typing of vertices in $gV(P_t)$ and have connections to data vertices in $dV(M_t)$ such that they comply with the data constraints imposed for vertices in $dV(P_t)$. This stage is implemented with two functions 5.2 and 5.3.
A second stage connects individual matched vertices in $gV(M_t)$ through edges that preserve the structure of the pattern indicated by P_t. These expansion steps are performed in a breath-first manner and they are the core iteration of the CP-E-PM algorithm to obtain the final results.

The result of CP-E-PM – after the second stage – is a set of subgraphs in M_t that identify instances of the pattern P_t. These subgraphs are stored in $\{PI\}$, where $|\{PI\}| = i$, with i indicating the number of edge-disjoint exact or partial instances of P_t in M_t. Overlapping instances can be found, but individual instances are indistinguishable, and therefore they are quantified as a single match.

An initial implementation of the CP-E-PM algorithm with basic - mostly syntactical - functions 5.2 and 5.3 was presented in [Gacitua-Decar 2008b]. A semantic enhancement was subsequently provided in [Gacitua-Decar 2009a].

Pseudocode of CP-E-PM. The pseudo-code of CP-E-PM is described in Table 5.1. CP-E-PM uses the ExactMatchTypes and ExactMatchAttributes functions to match types and attributes between model and pattern graph vertices. They are described in Table 5.2 and Table 5.3, respectively. A number of reduction steps modify the original adjacency matrix ($Adj(G_t)$), attribute matrix ($Attr(G_t)$), abstract type matrix ($AType(G_t)$) and data type matrix ($DType(G_t)$) of a recorded graph G_t into smaller matrices that contain only information related to vertices involved in the match. These matrices are $\widehat{AType}(M_t)$, $\widehat{Adj}(M_t)$, $\widehat{Attr}(M_t)$ and $\widehat{DType}(M_t)$. Through the

description of the CP-E-PM's pseudocode, it is explained how these matrices are obtained.

Table 5.1: CP-E-PM: Complete and Partial - Exact - Pattern Matching Algorithm.

CP-E-PM Algorithm

Input: $AType(M_t)$, $AType(P_t)$, $Attr(M_t)$, $Attr(P_t)$, $Adj(M_t)$, $Adj(P_t)$, $DType(M_t)$ and $DType(P_t)$.
Output: $\{PI\}$, where PI_i is the ith instance of P in M, centered on a vertex indexed by i in $\widehat{Adj}(M_t)$

```
1 : initiate {P_init}
2 : initiate {PI}
3 : ExactMatchTypes
4 : ExactMatchAttributes
5 : For each tmpP(m) in {PI} do
6 :   Do while ExpansionCondition == true
7 :     For each vertex u ∈ tmpP(m) indexed by i in Adj^(M_t) do
8 :       N_tmpP(m)(u) ← NeighboursOf(i in Adj^(M_t))
9 :       N_P_t(em(u)) ← NeighboursOf(j in Adj(P_t)), with j the index of em(u) in Adj(P_t)
10 :      If ⟨[em(N_tmpP(m)(u)) = N_P_t(em(u))] ∨ [N_tmpP(m)(u) ∉ tmpP(m)]⟩ == true then
11 :        Expand tmpP(m) with N_tmpP(m)(u)
12 :        ExpansionCondition ← true
13 :      Else
14 :        ExpansionCondition ← false
15 :      end if
16 :    end for
17 :  end do while
18 : end for
```

Stage 1: Exact Vertex Matching in CP-E-PM. The initial stage of CP-E-PM generates a set of initial temporal matches composed of single vertices in $gV(M_t)$. That set is denoted by $\{P_{init}\}$ and its elements are instances of vertices in $gV(P_t)$, i.e. pattern role instances. $\{P_{init}\}$ is expanded in the second stage to obtain the final set $\{P_i\}$ that contains the complete/partial instances of P_t in M_t. Various reduction steps are performed at this stage to improve the efficiency of the matching process. This first stage for exact vertex matching can be described as follows.

- First, each vertex u in $gV(M_t)$ (identified by the row index k in $AType(M_t)$) is related according to a type preserving relation to a vertex v in $gV(P_t)$ (identified by the row index i in $AType(P_t)$) - see Table 5.2. Type preservation is reviewed for each pair of no null entries (k, l) in $AType(M_t)$ and (i, j) in $AType(P_t)$. If abstract types indicated by j and l are equivalent, then the vertex indexed by k is considered an exact match - only by type - of the vertex indexed by i. That results in a match at type level between the vertices u and v.

Table 5.2: ExactMatchTypes identifies exact matches of abstract types for vertices in M and P and reduce matrices associated to M.

ExactMatchTypes

Input: $AType(M_t)$, $Attr(M_t)$, $Adj(M_t)$, $AType(P_t)$, $Attr(P_t)$, $\{P_{init}\}$
Output: updated $\{P_{init}\}$ and $\widehat{AType}(M_t)$, $\widehat{Attr}(M_t)$, $\widehat{Adj}(M_t)$

```
1 : For each non null entry (i, j) in AType(P_t) do
2 :   For each non null entry (k, l) in AType(M_t) do
3 :     If j = l then
4 :       match by type vertices indexed by k and i
5 :       add vertex indexed by k to {P_init}
6 :     end if
7 :   end for
8 : end for
9 : For each row k in AType(M_t) do
10 :  If the vertex indexed by k ∉ {P_init} then
11 :    ÂType(M_t) ← Reduce(AType(M_t), k) (reduction by rows)
12 :    Âttr(M_t) ← Reduce(Attr(M_t), k) (reduction by rows)
13 :    Âdj(M_t) ← Reduce(Adj(M_t), k) (reduction by rows)
14 :  end if
15 : end for
16 : For each column indexed by α_x in Âttr(M_t) do
17 :  For each column indexed by α_y in Attr(P_t) do
18 :    If α_x != α_y then
19 :      Âttr(M_t) ← Reduce(Âttr(M_t), α_x) (reduction by columns)
19 :      D̂type(M_t) ← Reduce(Dtype(M_t), α_x) (reduction by rows)
20 :    end if
21 :  end for
22 : end for
```

All vertices from $gV(M_t)$ that were matched by type are added to $\{P_{init}\}$. At the end of this step, each vertex from $\{P_{init}\}$ becomes a potential exact match of a pattern role in $gV(P_t)$. These vertices are reviewed later to check attribute level constraints satisfaction defined by pattern roles.

- A set of reduction steps eliminates all rows in $AType(M_t)$, $Attr(M_t)$ and $Adj(M_t)$ referring to vertices in $gV(M_t)$ which were not matched in the previous step. Additionally, $Attr(M_t)$ is further reduced by eliminating all columns indicating attributes that are exclusive to the previously eliminated rows, i.e., columns which have all row entries equal to zero in the reduced $Attr(M_t)$. The same is done for the matrix $DType(M_t)$. The resultant matrices are $\widehat{AType}(M_t)$, $\widehat{Adj}(M_t)$, $\widehat{Attr}(M_t)$ and $\widehat{DType}(M_t)$. These reduction steps reduce time of matching at attribute level.

- From previous graph vertices in $gV(M_t)$ matched by type, all their adjacent

Table 5.3: ExactMatchAttributes identifies exact matches of data vertices associated to attributes of graph vertices in P and M. $tmpP(m)$ is an initial temporal match of P in M centered in m.

ExactMatchAttributes

Input: $\widehat{Attr}(M_t)$, $Attr(P_t)$, $\widehat{Dtype}(M_t)$, $DType(P_t)$, $\{P_{init}\}$, $\{PI\}$
Output: updated $\{PI\}$

```
1 : For each non null entry (m, α_n) in Attr(P_t) do
2 :   For each non null entry (q, α_r) in Attr^(M_t) do
3 :     If vertices indexed by m and q are matched by type then
4 :       For each non null entry (α_r, δ_y) in row α_r from DType(P_t) do
5 :         For each non null entry (α_n, δ_x) in row α_n from Dtype^(M_t) do
6 :           If ⟨[(q, α_r) satisfies Constr_P : (m, α_n)] ∨ [δ_x = δ_y]⟩ then
7 :             match by (attribute, Constr_P) data vertices indicated by (m, α_n) and (q, α_r)
8 :             match by data type the attributes indicated by α_n and α_r
9 :           Else
10 :            eliminate vertex indexed by m from {P_init}
11 :          end if
12 :        end for
13 :      end for
14 :    end if
15 :  end for
16 : end for
17 : For each vertex m in {P_init} do
18 :  tmpP(m) ← m, with tmpP(m) ⊂ P_x and P_x ∈ {P_i}
19 : end for
```

data vertices are compared to data vertices in $dV(P_t)$ (see Table 5.3). The comparison aims to find matches at attribute level. An exact match at attribute level between a vertex u in $gV(M_t)$ and a vertex v in $gV(P_t)$ requires that all data vertices adjacent to u and connected through common attributes between u and v comply with the constraints imposed by data vertices adjacent to v. Constraints on data values associated to attributes of v provide a range of possible data values for attributes of u. A match at attribute level also verifies equivalence of data types by comparing data from $\widehat{DType}(M_t)$ and $DType(P_t)$ matrices. The set $\{P_{init}\}$ is updated to contain only vertices that have been matched at both levels: type and attribute.

Consider Figure 5.6 as a simplified example to illustrate Stage 1. A pattern and a model in different stages of the matching process are shown. In the first stage (next to the pattern at the left-hand side of Figure 5.6) seven vertices of the model were matched according to their equivalence to types and attributes of pattern vertices. Consider that the letter and number indicated in each vertex define the types and attributes for vertices. The pattern contains three vertices of type T and one vertex of

type G. Attributes are distinguished by a number. After matching vertices at type-level, all vertices in the model except $S1$ are matched according to their attributes. The result of matching at attribute-level contains all vertices of type T and numbers $4, 7, 9$; and vertices of type G and number 3. They are the only remaining vertices that would be expanded in the next stage. In real examples, the technique should consider the semantics involved in matching complex vertex types and attributes. Section 5.5 will discuss this subject in more detail.

Stage 2: Expansion Steps in CP-E-PM. The first stage of the algorithm provides a set of vertices ($\{P_{init}\}$) that correspond to exact instances of pattern roles (and connectors) from P. Vertices in $\{P_{init}\}$ are not connected and, therefore, partial and complete instances of the pattern P have not been identified yet. The second stage of CP-E-PM expands pattern role instances with neighbour instances. Figure 5.6 illustrates the expansion steps from an initial set of matched graph vertices to final pattern instances identified in a model graph. A pattern graph is shown at the left of the figure. For the sake of simplicity, data vertices and attributes are not shown. The same label for a pattern vertex and model vertex indicates equivalence at type and attribute level. The first expansion step extends single matched vertices with neighbours such that these neighbours are mapped by *ecm* (or *epc*), indicating a homomorphic relation between vertices (and edges) from model and pattern graphs. This first expansion step in Figure 5.6 is illustrated by the extension of vertices labelled by $T4$ to neighbours having a $T7$ label. The same relation holds in the pattern. The second and third expansion steps expand the subgraph (temporal pattern instance) at the right composed by vertices labelled with $T4$, $T7$ and the edge connecting them, with the edges and vertices labelled with $T8$ and $G3$. The result of the three expansion steps are two partial matches (containing a single vertex and two vertices) and one complete match composed of four vertices (the subgraph indicated at the right of Figure 5.6).

Expansion steps are repeated until all vertices in $\{P_{init}\}$ are visited.

Note that several exact or partial instances of a pattern graph P in graph model M might exist, and hence there are possibilities of finding overlaps between instances. CP-E-PM identifies overlapping instances as single subgraphs. The number of vertices in each of these subgraphs is the total number of vertices in the overlap plus all other vertices that are part of the instances but not the overlap.

Also note that in order to consider the directionality of edges in graphs representing concrete models and patterns, the algorithm uses the recorded graphs M_t and P_t, which are undirected versions of M and P. Figure 5.7 shows an example where two patterns that are exactly the same, except the direction of two arcs (directed edges), are matched over the undirected versions of the model graph M. The

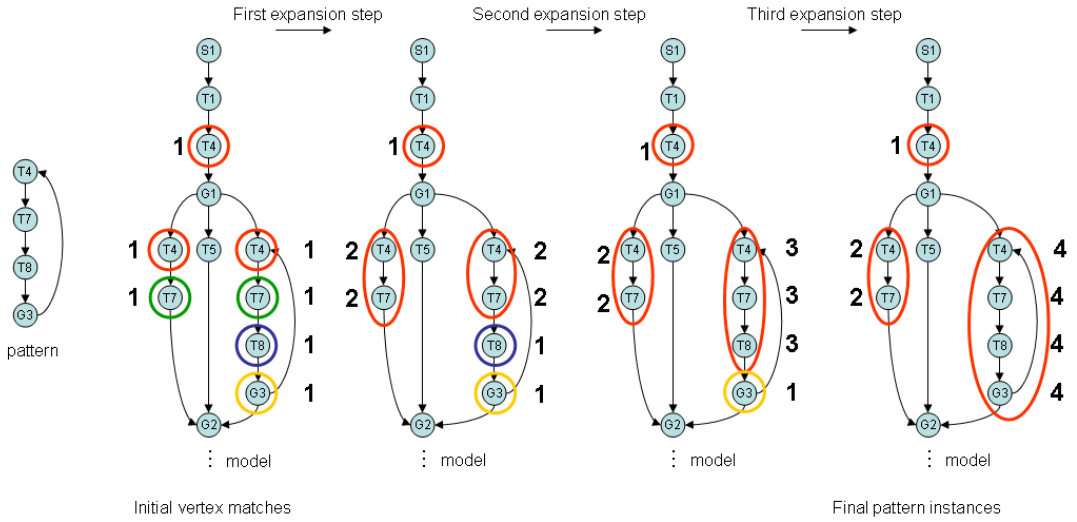


Figure 5.6: Illustration of expansion steps during exact pattern matching.

matched patterns are P and P'. P is the same pattern from Figure 5.6. P' is identical to P except that the direction of two arcs was interchanged. An exact and complete match of P in M is no longer a complete match in the case of P'. Two partial matches are found for P', one containing vertices labelled by $T4$ and the other one containing vertices labelled by $T7$, $T8$, $G3$ and the intermediate vertices labelled by $T7T8$ and $T8G3$. These latter vertices represent original edges in M.

Expansion steps performed during the second stage of CP-E-PM correspond to the detection of a homomorphism between P and a subgraph of M. Such a subgraph is a complete or partial instance of P in M, and it can be derived from a surjective or bijective local homomorphism (see Section 4.5 for details).

According to [Fiala 2008], for a connected simple graph H, the problem of detecting a locally surjective homomorphism between an arbitrary graph and H is solvable in polynomial time if and only if H has at most two vertices. In all other cases the problem is NP-complete. The complexity of the graph matching problem creates issues related to performance of the algorithm. Reduction steps and the way expansion steps are performed in CP-E-PM contribute to reduce the processing time of the algorithm. Section 8 describes the results of an empirical evaluation of the CP-E-PM algorithm. The results indicate that the algorithm performs in polynomial time - with complexity that is quadratic to the size of the (recorded) graph. These results are better or comparable to several algorithms in the literature (see a discussion in the related work section of Chapter 10).

On the other hand, in terms of processing several patterns over one or more target graphs, scalability could be addressed with an extended version of the algorithm that allows parallel processing of different patterns in a number of models.

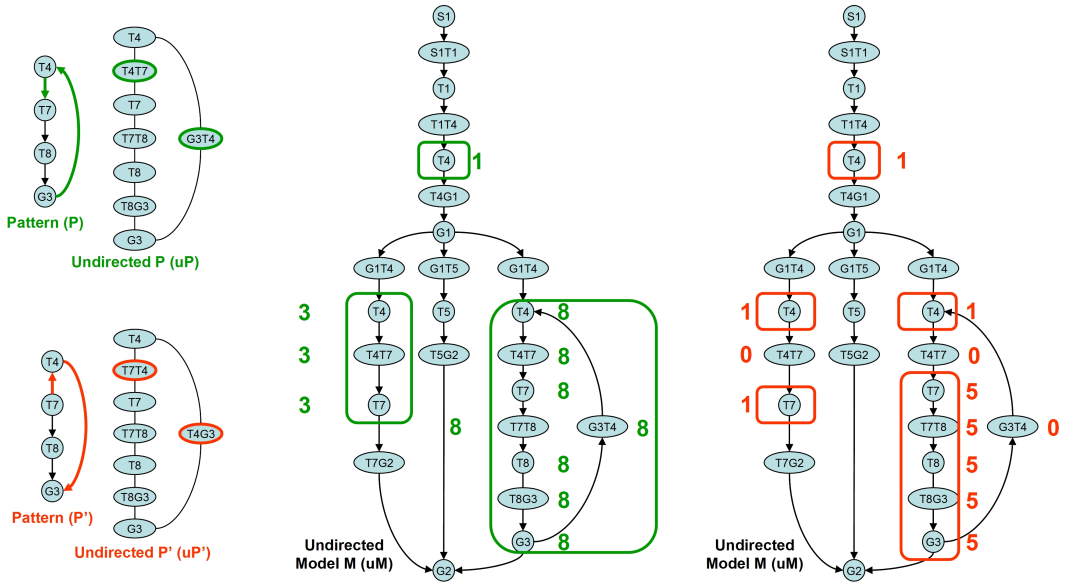


Figure 5.7: Matching over derived undirected graphs.

5.3.3 Complete/Partial & Inexact - Pattern Matching Algorithm (CP-I-PM)

In practice, finding *exact* (complete or partial) instances of a pattern is sometimes not possible. In a number of scenarios, pattern and model graphs could be subject to variability due a number of reasons, for instance, different implementations of pattern configurations, errors during the acquisition or documentation of processes and patterns, presence of nondeterministic elements when translating process-centric models to their graph representations, among other reasons [Gallagher 2006b], [Conte 2004].

Inexact process pattern matching aims to make more flexible the exact matching by allowing inaccuracies in observed graphs. That relaxes, to some extent, the exact matching of individual vertices at type and attribute levels. Additionally, expansion steps in CP-E-PM could relax the homomorphic condition between a pattern and its instances. These three kinds of relaxations can give rise to three flavors of an inexact pattern matching algorithm.

The main algorithm is denoted by CP-I-PM and covers the finding of complete and partial inexact instances. The three flavors are denoted by CP-I-Type-PM, that relaxes type matching and it means to allow the inclusion of subtypes as valid matches; CP-I-Attr-PM, that relaxes attribute matching by including vertices that do not satisfy the constraints imposed by pattern roles, however they are "close" to satisfy them; and CP-I-Strc-PM, that makes more flexible the structural preserving mapping considered by CP-E-PM. CP-I-Strc-PM would allow intermediate graph vertices that are not pattern role instances but that can be part of an inexact pattern instance because

they do not affect the intention of the pattern.

5.3.3.1 CP-I-PM with Relaxed Type Matching (CP-I-Type-PM)

Consider the model M and pattern P over ATM and ATP, respectively. Types of graph vertices from M and P are defined by the attributed type graphs ATM and ATP. In particular, types belong to the subsets $gV(ATM)$ and $gV(ATP)$.

For an *exact* matching using CP-E-PM, the function ExactMatchTypes considers a vertex u in $gV(M_t)$ a match of a vertex v in $gV(P_t)$ if their *types are equal*. In CP-I-Type-PM this condition is relaxed by allowing (1) the type of u be a subtype of v's type or (2) the type of v be a subtype of u's, but with the condition that constraints over u capture the restrictions defined by the type of v. This relaxed type of matching is performed by the function InexactMatchTypes. This function replaces ExactMatchTypes in line 3 of CP-E-PM (see Table 5.1). This change defines the difference between the CP-I-Type-PM and CP-E-PM algorithms. The pseudocode of InexactMatchTypes is described in Table 5.4.

Figure 5.8 illustrates an example that shows the two instances of the pattern $P1$ in the model from Figure 5.3. Instances of $P1$ - only considering vertex types - can be identified using a relaxed type matching. Subtypes of vertices in $P1$, defined at attributed type graph level, can derive a number of patterns that contain vertices whose types are subtypes of $P1$'s vertices types. Instances of these derived vertices are instances of $P1$ that were matched by relaxing the pattern vertices types. In this case, the *send/receive message* pattern roles have associated subtypes *receive/send bank transaction message* and *receive/send accounting message*. Other subtypes, for instance, *secure* and *unsecure receive/send bank transaction message* could be also defined. Instances of $P1$-*bt* and $P1$-*a* in Figure 5.8 are associated to *Receive/Send bank transfer order* and *Receive/Send settlement information* activities at process model level. Attribute matching could further filter out instances of $P1$.

A higher level graph relating possible pattern variations could be derived from the (attributed) type graph for pattern and model graph elements. Pattern variation (and combination) are only introduced in this work, a detailed investigation can be considered as future work.

5.3.3.2 CP-I-PM with Relaxed Attribute Matching (CP-I-Attr-PM)

The function ExactMatchAttributes used by the CP-E-PM algorithm allows data vertices in $dV(M_t)$ to be compared with data vertices in $dV(P_t)$. Compared data vertices are those adjacent to vertices in $gV(M_t)$, $gV(P_t)$ which were previously matched by type. An *exact* match at attribute level would relate a data vertex u_1 in $dV(M_t)$ to a data vertex v_1 in $dV(P_t)$ only if they are connected through the same type of attribute

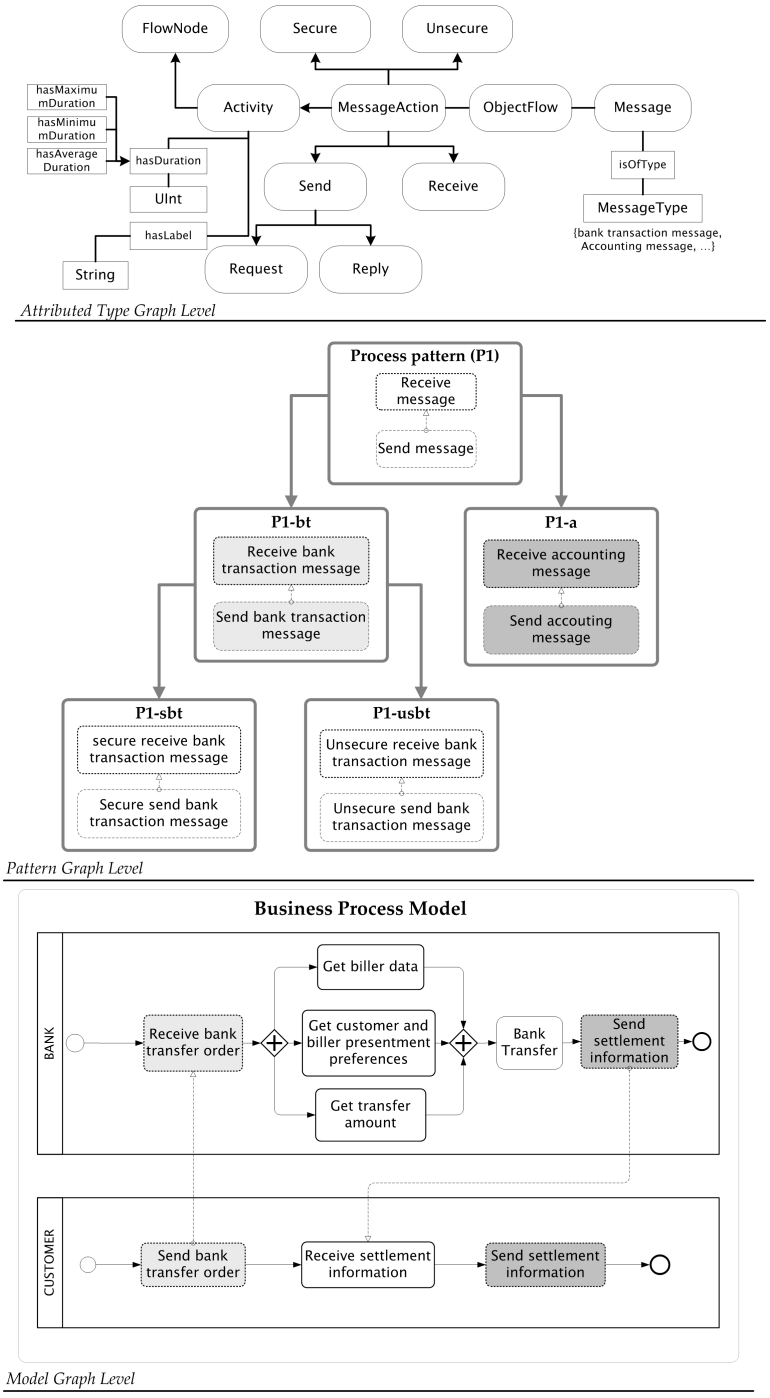


Figure 5.8: Illustration of relaxed type matching.

Table 5.4: InexactMatchTypes identifies inexact matches of abstract types for vertices in M and P and reduces matrices associated to M.

InexactMatchTypes

Input: $AType(M_t)$, $Attr(M_t)$, $Adj(M_t)$, $AType(P_t)$, $Attr(P_t)$, $\{P_{init}\}$, $Adj(ATM_t)$, $Attr(ATM_t)$, $Adj(ATP_t)$, $Attr(ATP_t)$

Output: updated $\{P_{init}\}$ and $\widehat{AType}(M_t)$, $\widehat{Attr}(M_t)$, $\widehat{Adj}(M_t)$

1 : **For each** non null entry (i, j) in $AType(P_t)$ **do**
2 : **For each** non null entry (k, l) in $AType(M_t)$ **do**
3 : **If** $\langle (l$ isSubtype of $j) \wedge [(j$ isSubtype of $l) \vee (l$ satisfies $Constr_P(j))]\rangle$ **then**
4 : match by type vertices indexed by k and i
5 : add vertex indexed by k to $\{P_{init}\}$
6 : **end if**
7 : **end for**
8 : **end for**
9 : **For each** row k in $AType(M_t)$ **do**
10 : **If** the vertex indexed by $k \notin \{P_{init}\}$ **then**
11 : $\widehat{AType}(M_t) \leftarrow$ Reduce$(AType(M_t), k)$ (reduction by rows)
12 : $\widehat{Attr}(M_t) \leftarrow$ Reduce$(Attr(M_t), k)$ (reduction by rows)
13 : $\widehat{Adj}(M_t) \leftarrow$ Reduce$(Adj(M_t), k)$ (reduction by rows)
14 : **end if**
15 : **end for**
16 : **For each** column indexed by α_x in $\widehat{Attr}(M_t)$ **do**
17 : **For each** column indexed by α_y in $Attr(P_t)$ **do**
18 : **If** α_x != α_y **then**
19 : $\widehat{Attr}(M_t) \leftarrow$ Reduce$(\widehat{Attr}(M_t), \alpha_x)$ (reduction by columns)
19 : $\widehat{Dtype}(M_t) \leftarrow$ Reduce$(Dtype(M_t), \alpha_x)$ (reduction by rows)
20 : **end if**
21 : **end for**
22 : **end for**

α ($\alpha \in aE(ATM) \vee \alpha \in aE(ATP)$) and all constraints imposed on v_1 are satisfied by u_1. In the case of *inexact* matching at attribute level, a match between a vertex u_2 in $dV(M_t)$ and a vertex v_2 in $dV(P_t)$ could exist, even though associated attributes are not equal and constraints imposed on v_2 are not entirely satisfied by u_2.

Inexact matching extends *exact* matching allowing mappings between semantically similar vertices and relaxing constraints defined in pattern graph attributes. Constraints are relaxed via a threshold allowing a range of additional values around an original constraint. A constraint c for data values associated to an attribute α of u, denoted by $c : (u, \alpha)$, is relaxed via the threshold τ, denoted by $\hat{c} = (c, \tau) : (u, \alpha)$. Similar to the algorithm CP-E-PM, that uses the function ExactMatchAttributes to match vertices at attribute level, CP-I-Attr-PM uses the function InexactMatchAttributes to *inexactly* match vertices at attribute level. The pseudocode of InexactMatchAttributes is provided in Table 5.5. The following two examples illustrate cases

of inexact matching of vertices at attribute level.

Example 1. Suppose two vertices $u_1 \in gV(M_t)$ and $u_2 \in gV(M_t)$ that correspond to *Send bank transfer order* (u_1) and *Send settlement information* (u_2) activities in Figure 5.8 and u_1, u_2 were previously matched by type to *Send bank transaction message* (v_1) and *Send accounting message* (v_2) in $P1$-*bt* and $P1$-*a*. v_1 and v_2 are associated by a subtyping relation to a vertex v of $P1$ ($v \in gV(P1_t)$) that correspond to the *Send message* activity. Also, suppose that u_1, u_2 have the adjacent attributes: *Label*, *averageCost*, *averageDuration* and *maximumDuration*; and v has adjacent attributes *Label* and *Duration*. Assume that the attributed type graphs ATM and ATP defining abstract types for M and P were previously integrated into the type graph shown at the top of Figure 5.8, where the attributes *averageDuration* and *maximumDuration* in $aE(ATM)$ are subtypes of the *Duration* attribute in $aE(ATP)$. Consider the data vertices d_1 and d_2 are connected to u_1 and u_2 through *averageDuration* and *maximumDuration*, respectively. A value of $d_1 = 12[minutes]$ connected to u_1 indicates that the *Send bank transfer order* activity has an average duration of $12[minutes]$. A value of $d_2 = 30[minutes]$ connected to u_2 indicates that the *Send settlement information* activity has an maximum duration of $30[minutes]$. Also, consider $c1 : d_I \leq 10[minutes]$, with $c_1 \in Constr_P1$ and d_I a data vertex in $dV(P1_t)$ defining the constraints for data values associated to the *Duration* attribute. The constraint $c1$ - for a data vertex adjacent to *Duration* and the vertex v - indicates that a message should be sent in at a time period $\leq 10[minutes]$. A threshold $\tau = 3[minutes]$ defines a relaxation boundary to allow a less strict matching between data vertices in $dV(M_t)$ adjacent to attributes *averageDuration* and *maximumDuration* to $d_I \in dV(P1_t)$. c_1 is relaxed through τ defining the relaxed constraint $\tilde{c}_1$, with $\tilde{c}_1 : d_I \leq 10 \pm \tau[minutes]$. Note that τ is expressed using the same data type of d_I. d_1 and d_2 do not satisfy c_1 and therefore they are not exact matches of d_I. However, if d_1 and d_2 satisfy $\tilde{c}_1$, they could be considered inexact matches of d_I. In this example, while d_1 satisfies $\tilde{c}_1$ ($d_1 < (10 + \tau)$ with $\tau = 3$), d_2 does not meet the restrictions imposed by either c_1 or $\tilde{c}_1$. Given the relaxed constraint $\tilde{c}_1$, since only d_1 can be matched to d_I, only the *Send bank transfer order* activity could be considered an instance of *Send message* from $P1$.

Example 2. Consider a Model (M), a Pattern (P) and two data vertices $d_M \in dV(M_t)$ and $d_P \in dV(P_t)$ connected through a *Label* attribute to $u \in gV(M_t)$ and $v \in gV(P_t)$, respectively. The data type for data values associated to d_M and d_P is *String*. d_M and d_P identify the labels *email client data* and *send customer information*, respectively. When the two previous labels are compared by a human, intuitively, s/he would match d_M and d_P. Data values comparison in **Example 1** considers rela-

tional operators such as $<, \leq$, which can be implemented with simple mathematical operators. However, other attributes such as the *Label* attribute can require more complex operators when checking constraints satisfaction and calculating similarity. In the case of an (automatic) *Label* matching, an implementation could rely on a completely syntactical comparison between the two labels, however it would not find a match between d_M and d_P in **Example 2**. An approach considering semantic similarity between the two sentences defining the labels would be more adequate than a pure syntactical approach. Semantic similarity calculation for vertex matching at attribute level is discussed in Section 5.5. This section only addresses the algorithmic method used by the function InexactMatchAttributes to compare attributes stored in the matrices representing the recorded graphs M_t and P_t.

Table 5.5: InexactMatchAttributes identifies inexact matches of data vertices associated to attributes of graph vertices in P and M. $tmpP(m)$ is an initial temporal match of P in M centred on m, with m in $\{P_{init}\}$ that contains the vertices previously matched by type.

InexactMatchAttributes

Input: $\widehat{Attr}(M_t)$, $(Attr(P_t), \{\tau\})$, $\widehat{Dtype}(M_t)$, $DType(P_t)$, $\{P_{init}\}$, $\{PI\}$
Output: updated $\{PI\}$

For each non null entry (m, α_n) in $Attr(P_t)$ **do**
  For each non null entry (q, α_r) in $\widehat{Attr}(M_t)$ **do**
    If vertices indexed by m and q are matched by type **then**
      For each non null entry (α_r, δ_y) in row α_r from $DType(P_t)$ **do**
        For each non null entry (α_n, δ_x) in row α_n from $\widehat{Dtype}(M_t)$ **do**
          If $\langle [(q, \alpha_r)$ satisfies $(Constr_P, \tau) : (m, \alpha_n)] \vee [\delta_x \cong \delta_y] \rangle$ **then**
            match by (attribute, $Constr_P$, τ) data vertices indicated by (m, α_n) and (q, α_r)
            match by data type the attributes indicated by α_n and α_r
          Else
            eliminate vertex indexed by m from $\{P_{init}\}$
          end if
        end for
      end for
    end if
  end for
end for
For each vertex m in $\{P_{init}\}$ **do**
  $tmpP(m) \leftarrow m$, with $tmpP(m) \subset P_x$ and $P_x \in \{P_i\}$
end for

5.3.3.3 CP-I-PM with Relaxed Structural Preserving Mapping (CP-I-Strc-PM)

Once graph vertices from $gV(M_t)$ to vertices in $gV(P_t)$ are matched at type and attribute level, a number of expansion steps are performed. In the case of aim-

ing to find exact structural matches, these expansion steps would link individual pattern role instances through existing edges in M_t. Added edges between individual matches (pattern role instances) are established only if neighbour pattern role instances satisfy certain structural conditions, i.e., they have to satisfy a structural preserving mapping with vertices (and edges) of the pattern graph P governed by specific graph homomorphisms.

Unlike exact structural matching, inexact structural matching considers mappings that allow other vertices beside the ones satisfying specific homomorphisms. These additional vertices are so-called *intermediate vertices*. Intermediate vertices are not any vertex on the path connecting exact pattern role instances. They must to satisfy certain conditions, which imply that found inexact pattern instances should not affect the intended result of applying the original pattern.

Pattern graphs make reference to configuration of vertices (pattern roles) that support the achievement of a desired result. This result can be expressed as an expected state of the objects being processed. The configuration of vertices defines how such an expected state is reached. Constraints defined by a pattern graph impose restrictions on how the expected state is reached. An inexact pattern instance should allow the achievement of the expected state and at the same time it should satisfy the constraints indicated in the pattern description. Describing an expected state in terms of the graphs M and P would require the introduction of additional elements used to model processes as graphs. In particular, a formal representation of the objects being processed. This section does not address this formal representation and it is limited to describe the algorithmic method to expand individual matches in CP-I-Strc-PM to identify *inexact* pattern instances. The description assumes the functionality for selecting appropriate *intermediate vertices* is provided elsewhere. The next section is focused on how to determine these appropriate intermediate vertices. These are used in CP-I-Strc-PM for connecting exact partial instances and to identify complete – and eventually partial – inexact pattern instances.

An example is introduced to facilitate the explanations in the next paragraphs.
Example: Suppose a process pattern P defines a common *document revision* process involving the repetition of two activities: *Analyse documents* and *Evaluate documents*, and a process model M contains a set of core iterative activities: *Analyse documents*, *Rank documents*, *Evaluate documents* and *Record revision cycle*. Figure 5.9 shows the model M and the pattern P. The core iterative activities in M resemble the pattern P, but not exactly. Intuitively, M contains an inexact pattern instance of P. The rest of the section describes how CP-I-Strc-PM could identify this inexact instance.

Expansion Steps in CP-I-Strc-PM. Consider the process model M, the pattern P and their recorded graphs M_t and P_t. The first stage of the CP-I-Strc-PM focuses

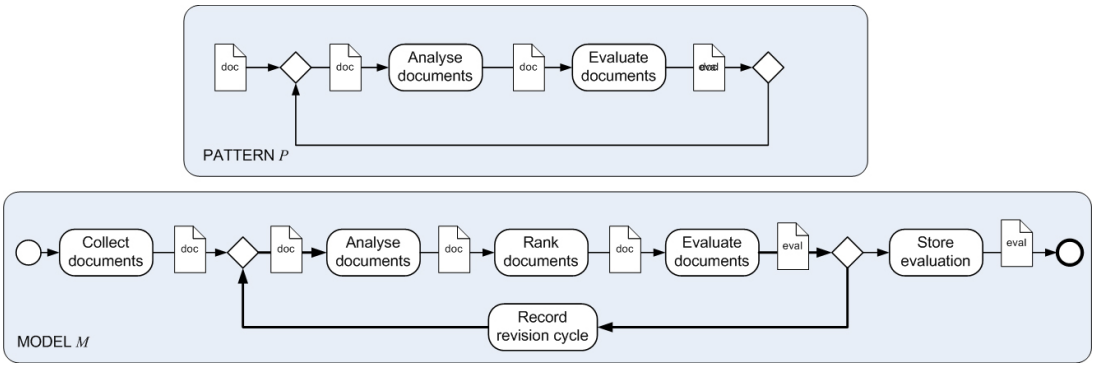


Figure 5.9: Example of intermediate vertices from an inexact pattern instance.

on finding individual pattern role instances by using the functions ExactMatchTypes or InexactMatchTypes and ExactMatchAttributes or InexactMatchAttributes, according to the end user requirements. The results generated by this first stage (individual pattern role instances) is the input of the next stage that performs a set of expansion steps. The expansion steps in CP-I-Strc-PM start in the same manner as CP-E-PM (Stage 2), but CP-I-Strc-PM goes beyond. It tries to connect exact isolated partial pattern instances to identify complete but inexact pattern instances. The connection is created through intermediate vertices and their adjacent edges.

Assume there is a function PartialInstanceConnector that takes the graph vertices from the boundaries of two partial exact pattern instances of P_T in M_t and returns a number of paths in M_t connecting the two instances. Each of the connecting paths starts in a boundary vertex from one of the partial pattern instances and ends in a boundary vertex of the other partial pattern instance.

The number of paths connecting two partial pattern instances would be determined by an end user. A single path connecting the two partial pattern instances is the minimum graph structure connecting two (edge-disjoint) partial exact instances. The maximum connecting structure would be defined by the amount of all outgoing edges of boundary vertices from the source partial pattern instance and the amount of all ingoing edges of boundary vertices from the target partial pattern instance.

Consider the pattern P and the model M in the example of Figure 5.9. Figure 5.10 illustrates the expansion steps used in CP-I-Strc-PM to find an inexact pattern instance of P in M. First, CP-I-Strc-PM identifies all pattern instance roles in M_t. Afterwards, it expands the individual matches with their neighbours[1] such that they preserve structure with respect to mapped vertices in P_t, i.e. CP-I-Strc-PM performs the same expansion steps as CP-E-PM in Stage 2. After all possible expansion steps

[1]Note that edges in M are vertices in M_t and therefore they are considered part of the possible neighbours to be added during expansion steps. In Figure 5.10, matched edges in M are highlighted with a wider line and red colour.

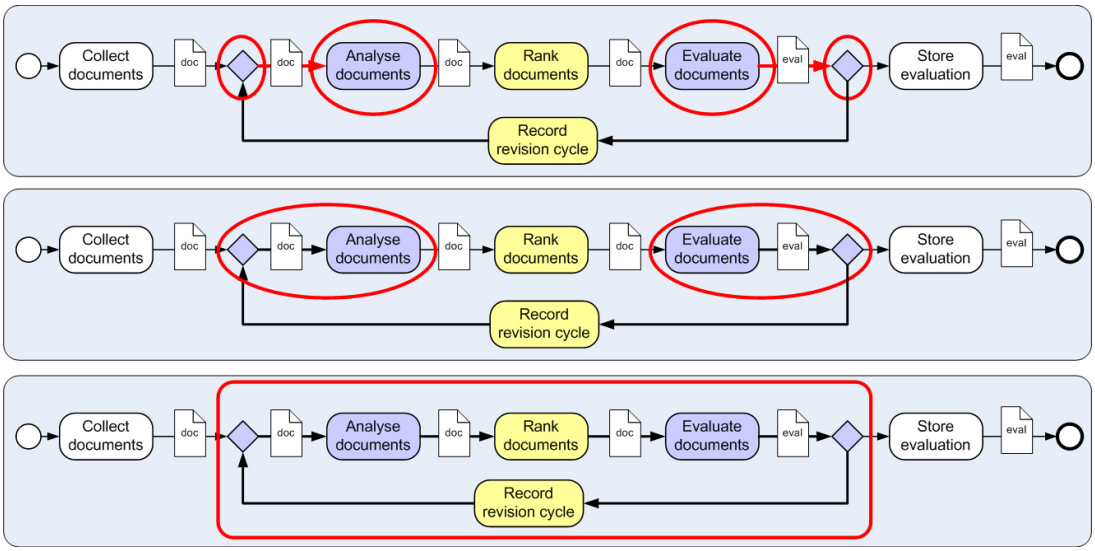


Figure 5.10: Illustration expansion steps during inexact structural pattern matching.

via exact structural matching are done, partial exact matches are obtained. The graph in the middle of Figure 5.10 identifies two exact partial instances of P in M. Consider these partial instances are referred as $P_x^{Partial}$ and $P_y^{Partial}$. The last expansion step of CP-I-Strc-PM aims to connect $P_x^{Partial}$ and $P_y^{Partial}$ to identify a complete structurally inexact instance of P in M. For that, CP-I-Strc-PM would use the PartialInstanceConnector function to identify possible paths connecting $P_x^{Partial}$ to $P_y^{Partial}$. Assuming the paths containing the *Rank documents* activity and the *Record revision cycle* activity (and adjacent edges) satisfy the constraints defined for edges connecting *Analyse documents* to *Evaluate documents* and the two control flows in P (see Figure 5.9), the result of all expansion steps in CP-I-Strc-PM would be as it is shown at the bottom of Figure 5.10. The framed subgraph is a structurally inexact instance of P in M.

Identification of Appropriate Intermediate Vertices for Inexact Structural Matching. Consider the activity identified as *Rank documents* in Figure 5.10. The *Rank documents* activity is considered an *intermediate vertex* from the structurally inexact instance of the *document revision* pattern. To be considered an intermediate vertex, *Rank documents* should not affect the results intended by the pattern. An obvious question is, how it can be determined whether an intermediate vertex is not affecting the intention of the pattern? To provide a possible answer, the graphs representing the process model and process pattern could be expanded to consider the information objects being processed. Representing not only processing steps, but also processed information objects, can allow the representation of expected results in terms of processed objects. Using the example in Figure 5.9, it is expected

that the intermediate vertices *Rank documents* and *Record revision cycle* activities do not transform the processed information object *doc* at the inputs of the activities *Analyse documents* and *Evaluate documents* in the model M. Also, the state of *doc* just before executing the *Analyse documents* or *Evaluate documents* activities has to satisfy their (pre)conditions. A detailed implementation of a technique to identify the appropriate intermediate vertices for inexact structural matching is out of the scope of this work, and it can be a included as a technical contribution for future work.

The different types of inexact matching reviewed in the last part of this section aims to provide a comprehensive framework for pattern matching, including situations where exact and complete pattern instances are only one type of instance that end users could be interested in.

5.4 Hierarchical Pattern Matching

Previous sections have addressed exact and inexact, partial and complete matching of process pattern instances in process model graphs. Model and pattern graphs in previous sections are flat process models. Flat models have elements without internal structure. Often, process descriptions have elements with internal structures. For instance, a composite activity in a process model can be disaggregated into a set of more granular connected process steps. Similarly, process patterns can refer to high level structures of connected patterns at lower layers of abstraction. Taking that into account, this section focuses on process pattern matching in hierarchically arranged process structures. In particular, the section addresses matching of process pattern graphs at higher abstraction layers based on edge-disjoint pattern instances at lower layers.

Consider a process model graph M and a set $\mathcal{P}$ of m process pattern graphs. Pattern graphs in $\mathcal{P}$ are hierarchically organised considering aggregation relations between pattern graphs. Each pattern graph P_k^i in $\mathcal{P}$ belongs to a certain level of abstraction i, with $i = 1, 2, \ldots, n$ and it is indexed by k, with $k = \{1, 2, \ldots, m\}$.

Matching patterns from the lowest layer of abstraction ($i = 1$) follows the same steps explained in previous sections. Any of the algorithms from CP-E-PM and CP-I-PM algorithm families can be used. The idea of hierarchical matching is that after pattern instances are found at the lowest level, if they are edge-disjoint, they are transformed to composite vertices using a lifting function to subsequently perform new a matching step at a higher layer of abstraction. Pattern instances at this higher level are formed by one or more *lifted* pattern instances from the previous lower layer. A sequence of matching-lifting steps is repeated until no more matches are found or no more lifting steps can be done. Thus, an algorithm to implement hierarchical

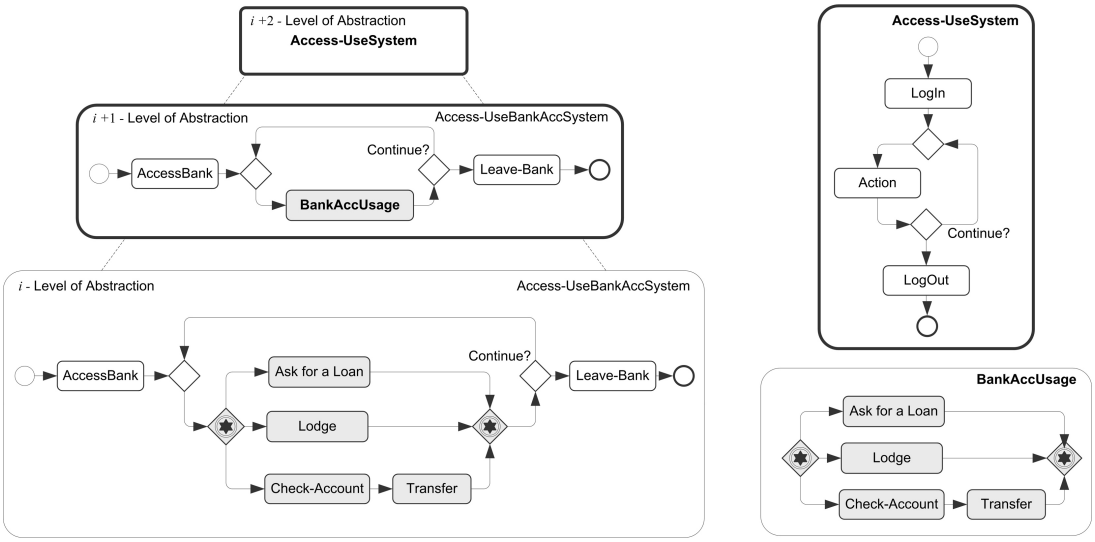


Figure 5.11: Hierarchical pattern matching.

pattern matching would iteratively match patterns at different abstraction layers. At one particular layer several patterns could be matched. A subsequent step would lift the model based on the found pattern instances. Subsequent matching-lifting steps would be perform at higher layers of abstraction. Steps would be performed until no more matching-lifting steps can be done.

Lifting was introduced in Section 4.5.1, which consists of a technique for abstracting a subgraph SG of a graph G in a single composite vertex v^{*^j}, where $*^j$ indicates that v is a composite vertex in the *jth* abstraction layer. Outgoing and ingoing edges from vertices in the boundaries of a lifted subgraph are the outgoing and ingoing edges of the composite vertex. For instance, consider the right-hand side in Figure 4.7, where two composite vertices PIz_1 and PIz_2 illustrate two lifted complete exact instances of the pattern Pz. Figure 5.11 illustrates another example where a model graph Access-UseBankAccSystem and the result of lifting it two times is shown. Two patterns, the BankAccUsage pattern and the Access-Use System pattern describe a set of common bank account usage activities and a typical (simplified) set of steps to access a generic system. The pattern BankAccUsage at layer i was matched and then the model lifted. Later the pattern Access-UseSystem was matched at layer $i+1$ and the last lifting step performed. That resulted in a model with a single composite vertex at layer $i+2$.

Attributes of composite vertices. Data values for some attributes of a composite vertex could be related to aggregated attribute values of vertices in its internal structure. For instance, consider a common attribute *Duration* for a sequence of activities representing the jth pattern instance of a pattern P in layer i, referred as PI^i_j. The

Table 5.6: H-PM Algorithm (Hierarchical - Pattern Matching Algorithm).

H-PM Algorithm

Input: Target Graph (M), set $\mathcal{P}$= $\{P_1, P_2, \ldots, P_m\}$ of m pattern graphs organised in i layers, selected pattern matching algorithm alg-PM.
Output: $\{PI\}$, where $PI^i_{x,k}$ is the xth instance of P_k in M at i layer, centred on a vertex indexed by x in $\widehat{Adj}(M^{*^i}_t)$ for layer i

```
1 :  Do while IterationCondition ∨ change == true
2 :   For each layer i do
3 :    For each pattern Pk ∈P do
4 :     If alg-PM (M*^i, Pk) != ∅ then
5 :      {PI}\k,i ← alg-PM (M*^i, Pk)
6 :      change ← true
7 :      For each pattern instance PI^i_x,k ∈ {PI}\k,i do
8 :       If isDisjoint(PI^i_x,k) == true then
9 :        M*^(i+1) ← lift(M*^i, PI^i_x,k)
10 :       If |M*^(i+1)| <= 1 then
11 :        IterationCondition ← false
12 :       end if
13 :      end if
14 :     end for
15 :    Else if
16 :     change ← false
17 :    end if
18 :   end for
19 :  end for
20 : end do while
```

value of the attribute *Duration* of a composite vertex $v^{*^{i+1}}_j$ representing the pattern instance PI^i_j in layer $i+1$ could be obtained by adding the individual durations of the sequence of activities in PI^i_j.

This simplified example illustrates one concern to be taken into account when implementing a lifting mechanism, i.e., a mechanism to (automatically) generate types and attributes for composite vertices. This feature strongly depends on the semantics of vertex descriptions that influences the ability of composing attributes and types. The implementation of a lifting function is out of the scope of this work. The reader might be interested in related work such as the *abstraction algorithm* presented in [Pinzger 2005] and the *lift transformation* in [Fahmy 2000].

Pseudocode of H-PM. The pseudocode of H-PM is described in Table 5.6. H-PM takes as input a recorded flat process model graph M_t (host model), a set of recorded flat process pattern graphs $\mathcal{P}$, and a reference to an algorithm alg-PM that would be used during pattern matching in each abstraction layer. Any of the algorithms from

the CP-E-PM and CP-I-PM algorithm families can be used. A lifting function is used by H-PM but implemented somewhere else. H-PM starts matching patterns from $\mathcal{P}$ in layer $i = 1$ over the host model M using alg-PM. After pattern instances in $i = 1$ are identified in M (M^{*^1}), M^{*^1} can lifted to obtain the model M^{*^2}. This is done replacing all found edge-disjoint pattern instances in layer 1 with composite vertices $v_1^{*^1}, v_2^{*^1} \ldots, v_s^{*^1}$. These composite vertices represent the instantiated patterns PI_1^1 to PI_s^1, with s indicating the total number of edge-disjoint pattern instances in level 1. If there are overlapped pattern instances, then H-PM terminates indicating this condition (existence of overlaps). Each subsequent iteration takes a lifted host model M^{*^i} and the subset of pattern graphs in $\mathcal{P}$ from layer $i+1$ to perform a pattern matching step that transforms M^{*^i} into $M^{*^{i+1}}$. These iterations are repeated until no more pattern instances can be found or lifting steps can be performed.

5.5 Semantic Matching

Preservation of the structural constraints defined by a process pattern is one of the aspects that has to be satisfied during the identification of process pattern instances on concrete process models. In previous sections, structural aspects of pattern matching were formalised using the edge/vertex preserving relations defined by graph homomorphisms. However, during process pattern matching not only structural preservation is addressed, but also satisfaction of constraints indicated by pattern roles is a prerequisite for vertices in pattern instances.

Pattern roles specify constraints defining the characteristic properties of an abstract pattern element. Beyond structure preservation, elements of a concrete model have to satisfy these constraints to be considered pattern role instances. In this section, matching at vertex level between vertices from process pattern graphs and process model graphs is addressed. Matching is based on the semantic similarity of pattern and model graph vertices. Matching at vertex level is a previous step to structural matching. CP-E-PM and CP-I-PM algorithms use specific type and attribute matching functions for this purpose. Similarity measures used by these functions are explained in this section.

5.5.1 Semantic Vertex Matching

Let M and P be a process model graph and a process pattern graph, respectively. M is an attributed typed graph $M = \langle AM, am \rangle$ over ATM and P is an attributed typed graph $P = \langle AP, ap, Constr \rangle$ over ATP and a set of applicable constraints $Constr$. M_t and P_t are the associated recorded graphs. Beside checking structure preservation, a solution to find a match between a vertex $u \in V(M_t)$ and a vertex $v \in V(P_t)$ has

to check if u satisfies the constraints defined by the type, attributes and data value restrictions for attributes of v.

The proposed techniques, previously discussed in Section 5.2, involve exact and inexact matching of types and attributes of vertices.

Regarding matching at *type level*, CP-E-PM identifies a match only if the types of two compared vertices $u \in V(M_t)$ and $v \in V(P_t)$ are exactly the same. On the other hand, CP-I-PM relaxes exact matching by allowing the type of v subsumes the type u. To say that u and v have the same type or they are related by subsumption, CP-E-PM or CP-I-PM would compare the abstract types of u and v defined by vertices in ATM and ATP, respectively.

Often, patterns and models are documented at different times and by different entities, and therefore is is likely that $ATP \neq ATM$. In this case, being able to identify *similarities* between types and attributes could be of interest. While subsumption is defined by a hierarchical (vertical) relation, *similarity* would refer to a relation that identify how close two values are within a neighbourhood.

In this work, matching at type level is restricted to identifying equivalent types or types in a subsumption relation, and where the associated ATM and ATP correspond to the same attributed type graph. Matching based on vertex type *similarity* would require additional and complex tasks, including merging different attributed type graphs related to models and patterns. The latter is similar to the studied problem of ontology merging, e.g., [McGuinness 2000], [Noy 2003] and it is beyond the scope of this thesis.

Figure 5.12 illustrates a situation where two models and a pattern refer to different (attributed) type graphs. In this case, two models $M1$ and $M2$ have instances of a pattern P. An additional step to merge $ATM1$, $ATM2$ and ATP would be required before starting any matching activity. Merging of $ATM1$, $ATM2$ and ATP would provide a unified type graph that CP-E-PM or CP-I-PM could use to perform the matching activity.

Matching at *attribute vertex level* compares the attribute values between common attributes describing two vertices. It is assumed that the attributed type graphs associated to a model M and a pattern P would refer to a single common graph, say AT, where $ATM = ATP = AT$. Matching at *attribute vertex level* using CP-I-PM would expand the exact matches identified by CP-E-PM to inexact matches that identify *similar* attribute values for common attributes describing $u \in V(M_t)$ and $v \in V(P_t)$. Compared attribute values would be those referring to the same (abstract) attributes defined in AT. Attribute values would not necessarily satisfy a subsumption relation and they might not always be hierarchically organised.

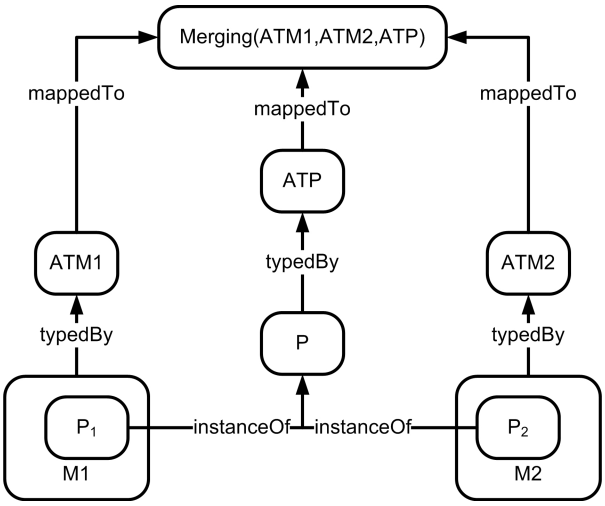


Figure 5.12: Merging of (attributed) type graph for inexact vertex matching by type.

5.5.2 Type Vertex Similarity

Consider M and P as above. Also, let the vertex $u \in V(M)$ be a potential pattern role instance of v in M, and the vertex $v \in V(P)$ be a pattern role from P. To be an instance of v, a necessary – even though not sufficient – condition for u is that the type of v has to be equal or a subsumer of the type of u. When the latter condition is true, it is said that there is a matching *at type vertex level* between u and v. To check this condition, the abstract types of u and v have to be related in a hierarchical structure, where the type of v would be at a higher level than the type of u.

Abstract types and attributes of v are defined by ATP according to the mapping $ap : v \rightarrow t_v$, with $t_v \in ATP$. Similar, abstract types and attributes of u are defined by ATM according to the mapping $am : u \rightarrow t_u$, with $t_u \in ATM$. A match between u and v at type vertex level indicates that t_v subsumes t_u.

For $ATP = ATM$ (referred as to AT), the function CP-I-PM would check if the depth of t_v in AT is equal or greater than the depth of t_u in AT. Figure 5.13 illustrates an example where the abstract type t_v subsumes the abstract type t_u in a graph AT, with $d(t_v) > d(t_u)$.

Note that $\varepsilon_1, \ldots, \varepsilon_7$ define *is-a* relations and the end of the relation identified with an arrow indicates a more abstract type.

5.5.3 Attribute Vertex Similarity

Once checked if the vertices u and v are matched at type vertex level, attributes of u and v are compared. In order to say that u is a match of v at attribute vertex level, the data values for attributes of u have to be - at least - similar to the data values of associated attributes of v. Associated attributes between v and u are those

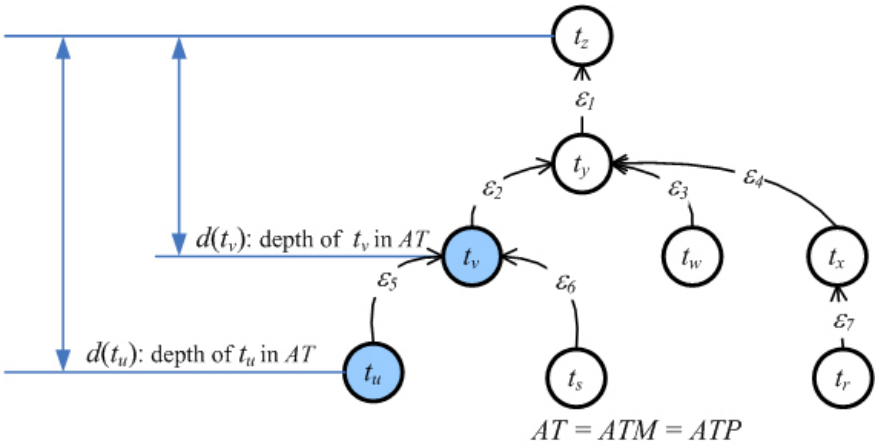


Figure 5.13: Example of type graph AT, where abstract type t_v subsumes the abstract type t_u.

referring to the same descriptive aspects. Ideally, the attributes defined for v are also attributes describing u. This can be true when $ATM = ATP = AT$ and u and v have the same type. Assuming a set of (abstract) attributes $\{\alpha_v\}$ defined in AT and used to describe v, and the same set of attributes being a subset of the attributes describing u, a match at attribute vertex level between u and v would be identified if the *similarities* between data values for $\{\alpha_v\}$ in u and v are within a specified threshold. This threshold identifies a region of similar data values for attributes describing the pattern role v and a subset of attributes of interest describing u. If data values of common attributes describing the model element u and the pattern role v are similar enough (within the range defined by the threshold), then the model element u could be considered a match at *attribute vertex level* of the pattern role v.

In order to calculate the similarity between data values for common attributes between the u and v vertices, the attributes and their values are organised in vectors. The attribute vectors of u and v are identified by $\vec{u}$ and $\vec{v}$, respectively. To compare both vectors and calculate their similarity, a weighted measure of distance $dis(\vec{u}, \vec{v})$ is calculated. Values for this distance measure range between zero and one. Similarity between the attribute vectors $\vec{u}$ and $\vec{v}$ would be one minus their distance.

Similarity between $\vec{u}$ and $\vec{v}$, $sim(\vec{u}, \vec{v})$, is calculated based on the formulation of the weighted Minkowski distance [Cha 2007] as,

$$sim(\vec{u}, \vec{v}) = 1 - \left(|\delta_i \cdot dis(\vec{u}_i, \vec{v}_i)|^p\right)^{1/p}, 1 \leq i \leq |\vec{v}| \tag{5.1}$$

Note that other attributes in $\vec{u}$ are deliberately omitted since they do not concern the process pattern configuration P. The choice of the attributes describing a process pattern is made by considering both, its intended application and also its potential

users.

Also, note that $dis(\vec{u}_i, \vec{v}_i)$ is the normalised *dissimilarity* between $\vec{u}$ and $\vec{v}$ in the attribute i. Values of dissimilarity range between 0 and 1, with 0 representing equality. Dissimilarity can become a *distance* measure, if distance is possible to calculate. According to the nature of each attribute, different measures of dissimilarity (or distance) can be considered. δ_i is a weighting factor to emphasize or deemphasize the ith attribute value. We assume attributes are independent. p determines the measure's norm. For $p = 2$, vertex similarity becomes a measure based on the Euclidean distance.

5.5.4 The Label Attribute and Label Similarity Calculation

One of the most common attributes of a process graph vertex is its label. Often, labels are *sentences* in natural language. Only a few approaches [Awad 2008b], [Dongen 2008], [Dijkman 2009b] have considered process element labels as part of the elements of comparison between process models.

In this work, to determine if the label of a process model element refers to a label of a process pattern role, the similarity between those two labels is calculated based on the *sentence similarity* measure described in [Li 2006]. This measure is convenient in the process pattern matching context since the elements required to evaluate the measure are dynamically generated using only the information from the words contained in the two labels. The measure considers the semantic similarity among words in the two sentences (labels), which is derived from a Lexical Knowledge Base (LKB) and a corpus, and the word order on the sentence meaning. LKBs are frequently organised as a hierarchy of words defining concepts (for example, WordNet [WordNet 2010] or other more specific LKBs targeting particular business domains). Semantic similarity between words is calculated based on the length of the path connecting the words in the hierarchy and their depth in it. By observing the direction (from bottom to top) of the path connecting two words in the hierarchy, it is possible to discriminate between abstraction or refinement of concepts. The latter can be used as indication that a vertex label is an abstraction of another vertex label.

Vertex label similarity calculation here is simplified by avoiding word disambiguation, abbreviations expansion and acronyms replacement. Word disambiguation requires the analysis of the context where the word appears [Navigli 2009].

Similarity between the vertex labels $\ell(u)$ and $\ell(v)$, where u is a vertex from a process model graph M and v is a vertex from a process pattern configuration P, is derived from the weighted sum of similarities between their associated lexical

semantic vectors and word order vectors,

$$sim_{label}(\ell(u), \ell(v)) = \rho \cdot sim(\vec{w}(u), \vec{w}(v)) + (1 - \rho) \cdot sim(\vec{o}(u), \vec{o}(v)) \tag{5.2}$$

The lexical semantic vectors $\vec{w}(u)$ and $\vec{w}(v)$ represent quantifiable values regarding the meaning of words in u and v's labels. The values are based on information from a lexical knowledge base and corpus. $\vec{o}(u)$ and $\vec{o}(v)$ represent quantifiable values regarding the word order in the sentences. ρ determines the relative contributions of the lexical semantic vector similarity and the word order vector similarity measures. If syntax is less relevant, according to [Li 2006], a value between 0.5 and 1 should be assigned to ρ.

5.5.4.1 Similarity Between Lexical Semantic Vectors

Similarity between the lexical semantic vectors $\vec{w}(u)$ and $\vec{w}(v)$ is defined as the cosine coefficient between them,

$$sim(\vec{w}(u), \vec{w}(v)) = \frac{\vec{w}(u) \cdot \vec{w}(v)}{\|\vec{w}(u)\| \|\vec{w}(v)\|} \tag{5.3}$$

$\vec{w}(u)$ and $\vec{w}(v)$ are vectors with m entries. m is the number of words in a joint word set W containing all the different words from the two labels $\ell(u)$ and $\ell(v)$, hence $m = |W|$. Each ith entry $\vec{w}_i(u)$ with $i = 1, ..., m$ is derived from evaluating the similarity between the word from the ith entry in the joint word set W, annotated $w_i(u)$, and the most similar word from the label $\ell(u)$, annotated $\tilde{w}_i(u)$. In turn, the value obtained from the word comparison is weighted by the individual information content of the two compared words,

$$\vec{w}_i(u) = sim_W(\tilde{w}_i(u), w_i(u)) \cdot I(\tilde{w}_i(u)) \cdot I(w_i(u)) \tag{5.4}$$

$I(w_i(u))$ and $I(\tilde{w}_i(u))$ refer to the information content of the words referred to $w_i(u)$ and $\tilde{w}_i(u)$. The information content of a word ($I(w)$) is derived from its probability (relative frequency) in a corpus. The higher the frequency of the word in a corpus, the less information it would contain.

$$I(w) = 1 - \frac{\log(n+1)}{\log(N+1)} \tag{5.5}$$

where n is the frequency of the word w in a corpus with N words.

In order to obtain the value of an entry in Equation (5.4), we need to calculate the similarity between two words (sim_W). We use the word similarity measure from

[Li 2003]. This measure, described in Equation (5.6), is a function of the path length (le) connecting the two words in the lexical knowledge base and the depth (de) of their common subsumer. The latter helps to differentiate the similarity between a pair of words that refer to more abstract concepts and the similarity between a pair of words that refer to more concrete concepts.

$$sim_W(w_1, w_2) = e^{-\alpha le} \cdot \frac{e^{\beta de} - e^{-\beta de}}{e^{\beta de} + e^{-\beta de}} \tag{5.6}$$

Adequate values for the constants α and β can be obtained experimentally depending on the used lexical knowledge network. In [Li 2003], $\alpha = 0.2$ and $\beta = 0.45$ are the proposed values for the lexical knowledge network WordNet. WordNet is also used in part of the evaluation section of this work - see Section 8.5 for more details.

5.5.4.2 Similarity Between Word Order Vectors

Similarity between two word order vectors $\vec{o}(u)$ and $\vec{o}(v)$ associated to the labels $\ell(u)$ and $\ell(v)$ is derived from their normalized difference,

$$sim(\vec{o}(u), \vec{o}(v)) = 1 - \frac{\|\vec{o}(u) - \vec{o}(v)\|}{\|\vec{o}(u) + \vec{o}(v)\|} \tag{5.7}$$

$\vec{o}(u)$ and $\vec{o}(v)$ are obtained from the order in which the words in $\ell(u)$ and $\ell(v)$ appear. The order is established based on a joint word order vector $\vec{O}$. $\vec{O}$ defines an order for words in the joint word set W used in Equations (5.3) and (5.4). If a word in $\ell(u)$ is in $\vec{O}$, the entry associated with that word in $\vec{o}(u)$ is its index in $\vec{O}$. If the word is not in $\vec{O}$, then two possible entries can be assigned. One is the index of the most similar word in $\vec{O}$ (only if the similarity between the compared words is greater than a threshold σ_O); otherwise, a value equal to zero is assigned to the entry.

Section 8.5 presents an example illustrating how this approach for semantic vertex similarity can be utilised to enhance the structural pattern matching approach presented in the previous sections.

5.6 Summary

In this chapter, a technical contribution involving a set of algorithms for a comprehensive framework for pattern matching was presented. A pattern matching technique for the LABAS framework (presented in Chapter 3) can use implementations of these algorithms to automate the pattern matching activity. Note that automation would include end user interaction to verify or modify, if necessary, the obtained

results. The algorithms include solutions to identify complete and partial pattern instances, and exact and inexact pattern instances. Hierarchical matching and semantic matching were also addressed. Inexact pattern matching was discussed in relation to inexact type matching, inexact attribute matching and inexact structural matching. Ideas for future work in the latter aspect were described. Hierarchical matching considered the possibility of patterns occurring at different levels of abstraction. This approach consisted of an iterative process that abstracts edge-disjoint pattern instances into composite vertices and iteratively matches patterns in higher abstraction levels. The iteration process required a graph lifting function that was not addressed in this work but can be adopted from other works. Semantic matching addressed the problem of vertices comparison for initial stages of the proposed pattern matching algorithms. Vertices types and attributes often have to be compared beyond their syntax. Comparison and matching of vertices at semantic level was fundamental for providing solutions that can be used with graphs representing realistic process-centric descriptions.

Chapter 6

Pattern Discovery

Contents

6.1 Motivation to a Pattern Discovery Solution

Among the pattern-based techniques in the proposed framework for SOA design and integration (Chapter 3), pattern discovery can be exploited to guide the definition of new services based on unknown and frequent process sections which can be documented and reused in the form of process patterns.

Previous sections focused on process pattern matching, whose aim is to identify known process patterns in process models. The motivation behind it is to support automatic process pattern matching as an analysis instrument during the definition of process-centric services based on proven designs documented as process patterns. Process patterns can provide guidelines to design new (software) services that can be used during enterprise process and application integration projects. However, in a number of organisations process steps are already supported by existing software components. Identifying recurring connected process steps can provide an opportunity to define reusable services that can be implemented encapsulating existing software components. This idea is aligned with the basic principle of software reuse in SOA [Erl 2004] and it can support software component rationalisation [Albani 2006]. Considering this opportunity, this section refocuses the attention from process pattern matching to the discovery of frequently occurring substructures – that can be captured as patterns – on large-scale business process models. The pattern matching algorithms introduced in the previous chapter are used as the basis of the pattern discovery technique proposed in this chapter. Also, semantic variations and generalisation can potentially be used in this new scenario.

Pattern discovery scenarios. The problem of identifying recurring connected process steps in process models is addressed as a problem of frequent pattern discovery in graphs [Kuramochi 2005]. Two distinct scenarios can be considered in this regard: the graph-transaction scenario and the single-graph scenario. The former refers to the discovery of subgraphs that occur frequently across a set of input graphs (graph transactions repository). The result of an algorithm for discovering patterns in the graph-transaction scenario is a set of graphs containing the frequent subgraph (pattern) across graphs in the repository. A graph is considered part of the result irrespective of how many times the pattern occurs in a particular transaction. Instead, for the single-graph scenario, an algorithm would discover the subgraphs that occur multiple times in a single, large input graph. The problem formulation and the input data used by algorithms in these two scenarios have inherent differences. According to [Kuramochi 2005], the algorithms developed for the graph-transaction scenario cannot be used to solve the problem defined for the single-graph scenario, whereas algorithms for the latter scenario can be easily adapted to work in the graph-transaction scenario. The problem and solution presented in this section are defined on the basis of a single-graph setting scenario. Such a single graph represents an – often large and complex – process model. Discovered patterns from this process model graph would indicate potential reusable process-centric services. Trace links to (lower) application architecture levels - see traceability model in Section 3.4 - would link processes to software components that can be analysed with the aim of being rationalised.

6.1.1 Matching versus Discovering Patterns in Graphs

Although there are similarities between the pattern discovery problem and the pattern matching problem described in previous sections, the formal relationship between a pattern and its host graph is different for both problems. Pattern matching on graphs involves the identification of a homomorphic relation between a given pattern graph P and a given graph model M. Instead, pattern discovery only takes the graph model M as input and, in order to discover unknown (frequent) patterns in M, M is compared to itself during the search for reoccurring subgraphs. Just as the homomorphic mapping from P to M formalises the structural preservation relation between P and its pattern instances in M, an *endomorphic* relation from M to subgraphs of itself formalises the structural relation involved in the frequent pattern discovery problem.

Pattern Size and Occurrence Frequency. Discovering a frequent pattern in M involves the identification of a subgraph U that appears in M a specific number of

times that is considered *frequent*. The occurrence frequency of U in M and also the size of U are defined by end users. Users can be interested only in subgraphs occurring in M at least a specific number of times f_U. On the other hand, if a subgraph U is considered a subgraph (pattern) of interest, then so can be subgraphs of U. Determining what size of U is adequate depends on the final goal that triggered the discovery of patterns in a graph. The goal in this work is to define boundaries on process descriptions as guidelines to define new services. Those boundaries are identified with the purpose of benefiting service reuse across the process. Service reuse is strongly influenced by the service application scenario and therefore the input from end users (designers) is important. During the design of new services based on discovered patterns, end users would deal with a tradeoff between size and frequency of the discovered patterns. A greater occurrence frequency would benefit service reuse. On the other hand, a greater pattern size would lead to a coarser grained service, and indirectly, it could benefit the performance of a service composition created to automate or integrate a business process. In terms of performance, coarser grained services (as building blocks of service compositions) can be seen as beneficial compared to finer grained services. By composing finer grained services addressing the same integration problem, a larger number of services – therefore, more service requests and responses – would be involved. That increased number of requests and responses could cause undesired levels of performance due to the added overhead originated from the new and more complex service composition. Thus, pattern size and its occurrence frequency affects the decisions involved during the design of new pattern-based services. Methodological guidelines for how to adjust the parameters of an algorithm for pattern discovery and the implementation of the algorithm in a tool that end users can interact with would assist with semi-automated support to the design of the new services. The affected design guidelines are those related to service *reuse* and service *granularity*.

Counting occurrence frequency of a subgraph. There are different approaches to count the occurrence frequency of a subgraph U in M. If the subgraph is frequent enough, then it is considered a pattern of interest. Counting approaches vary according to how overlaps among of a subgraph U are considered – see Section 4.5.1 for more details on overlaps in graphs. One alternative would count all occurrences of a subgraph U, including those belonging to an overlap. Another alternative would count occurrences of U in M only if they are edge-disjoint (i.e., they do not share edges in M). Counting occurrences from overlaps could lead to the fact that the counting of a subgraph occurrence frequency does not decrease monotonically as a function of its length, causing a pattern discovery solution to become untractable [Vanetik 2002].

6.1.2 Frequent Pattern Discovery in Process Graphs

Consider a process model M, where M is a typed attributed graph $M = \langle AM, am \rangle$ over ATM. The problem of frequent pattern discovery in process graph concentrates on finding connected edge-disjoint subgraphs occurring in M. A subgraph U of M is considered a frequent pattern if it appears in M at least a number f of times, where f is the so-called occurrence frequency threshold.

In the case of an *overlap* containing recurrent edge-disjoint subgraphs, counting occurrences of the subgraph involves the calculation of an independent set of vertices from the overlap [Kuramochi 2005]. For a graph $H = (V, E)$, a set of vertices $I \subset V(H)$ is called *independent* if for every pair of vertices in I, the pair is not connected by an edge in $E(H)$. The independent set is called maximum if for every vertex v in I there is an edge in $E(H)$ that connects v to a vertex u in $V(H)$ but not in $V(I)$. Exact counting of the occurrence frequency of a pattern involves calculating the *exact maximum independent set* of an overlap containing the pattern. Because the calculation of the exact maximum independent set of graphs is NP-complete [Lawler 1980], an *approximate* pattern discovery would try to find *as many as possible* subgraphs with an occurrence frequency at least f. This approximate solution is used in many practical cases such as in [Kuramochi 2005], [Inokuchi 2005] and also in this work.

Motivating Example. Figure 6.1 shows an example of a business-level process-centric service composition extracted from [Rabhi 2007]. The process-centric service implements a *trading strategy simulation* process and it has highlighted - with borders coloured in red and blue - instances of frequent patterns. Examples of these patterns are shown in Figure 6.2. $P1$ is the larger frequent pattern occurring exactly two times in the process from Figure 6.1. $P2$ is a smaller pattern that occurs more frequently in the process, and it corresponds to a subgraph of the graph representing $P1$. On the other hand, $P3$, $P4$ and $P5$ are associated to subgraphs of the graph representing $P2$, and therefore $P1$. $P3$ and $P4$ consist of single elements that can be abstracted by the *Action on message* element, which in turn, it is a more concrete element that refines the *Action* element. The same situation can be considered for the *Process Interrupted?* element from $P5$, which refines the more abstract *Decision* element. If an algorithm for frequent pattern discovery allows inexact matching by relaxing the vertex matching to allow two elements to be matched if they are semantically similar but no exactly the same, it could be considered that the model from Figure 6.1 has five instances of $P1$ (two exact instances and three inexact instances that include semantically similar elements). Intuitively and considering the semantics of labels, the elements with borders coloured in red are more similar to the pattern elements in Figure 6.2 than the elements with borders coloured in blue. For instance, for the

trading strategy and *simulation* service (middle of Figure 6.1), the *Generate and Submit orders* activity is semantically similar to *Send message*. Also, the *Monitor Market Events* activity can be (semantically) associated to the abstract *Action* element in Figure 6.2. A threshold defining how similar should be considered two elements to say that they are instances of a same pattern role has to be defined by an end user. An algorithm for pattern discovery which is flexible could allow for this inexact matches. Also, the algorithm should allow end users to identify partial matches of a pattern of interest. For instance, after discovering that $P1$ occurs exactly two times in the model, end users may be interested in to know if there are partial instances of $P1$. In the case of the example, there are indeed. They are associated to the frequent patterns $P2$, $P3$, $P4$ and $P5$.

Note that the example here was chosen because it can fit in one page. Real processes can be larger and more complex, requiring a means to automate some of the analysis tasks that can be difficult and expensive to do without support.

6.2 Matching-based Algorithm for Pattern Discovery

This section describes a technique to find frequent patterns in an (often large) process model M. The technique is based on an algorithm that uses the pattern matching algorithm families from previous sections. The inputs to the pattern discovery algorithm - named λ-PD algorithm, where λ is any of the pattern matching algorithms from the CP-E-PM and CP-I-PM families described in the previous chapter - are:

- the recorded graph M_t of the target process model M,
- the maximum expected size of a subgraph U representing a pattern ($|V(U) + E(U)|$), and
- the occurrence frequency threshold f_{min} indicating the minimum number of times that U has to be contained in M. Since there is no interest in finding the trivial automorphism of M and it is expected a (frequent) pattern to occur at least two times in M, then $|V(U) + E(U)|$ is trivially bounded by $|V(M_t)|$ and $f_{min} \geq 2$.

The idea behind the proposed λ-PD algorithm is to create temporal patterns originated from each vertex in $V(M_t)$, subsequently expand them and then check if they occur at least the number of times defined by an occurrence frequency threshold f_{min}. Before any expansion step is performed, temporal patterns formed by single vertices are discarded early if they do not reach the occurrence frequency f_{min}. Expansion steps are performed until the maximum desired size of the pattern is reached or overlaps extending the target model graph are found. The pseudo code of λ-PD is described in Table 6.1 and explained in the rest of the section.

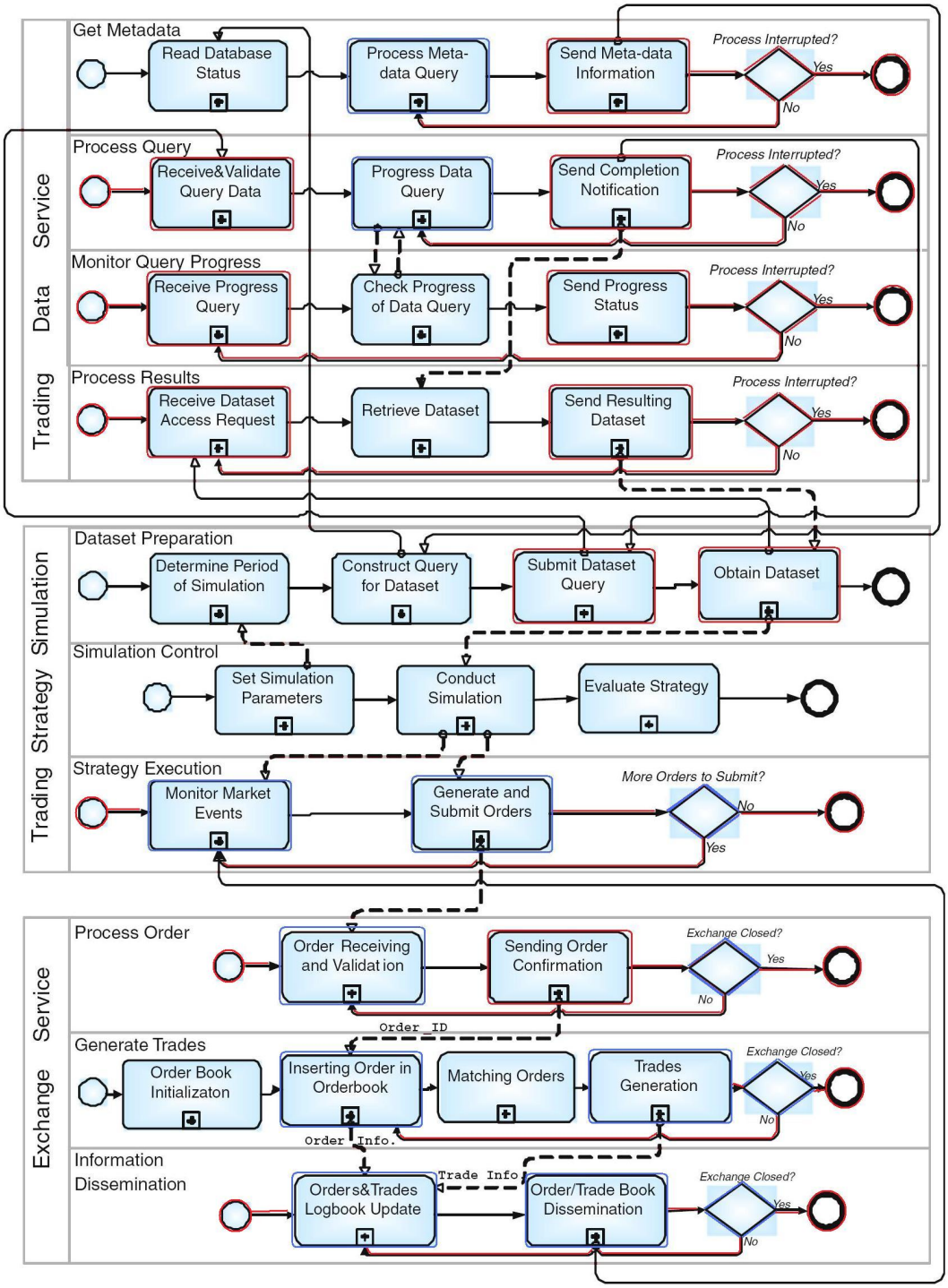


Figure 6.1: Example of an abstract process-centric service with frequent pattern instances.

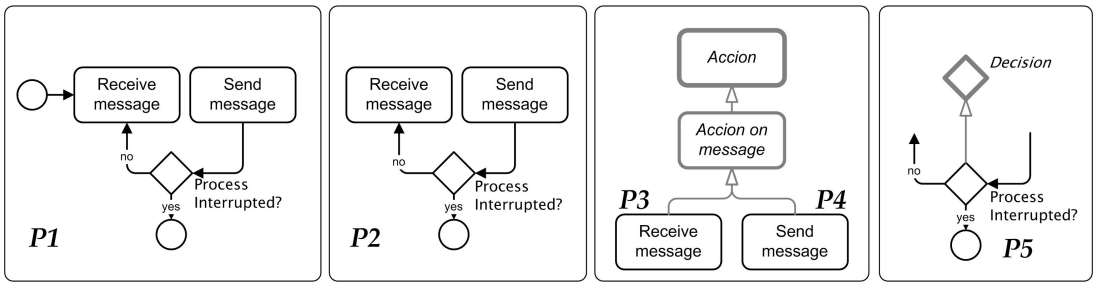


Figure 6.2: Example of an abstract process-centric service with frequent pattern instances.

Table 6.1: λ-PD Algorithm - Pattern Discovery Algorithm based on λ pattern matching algorithm, with λ among the CP-E-PM and CP-I-PM families.

λ-PD Algorithm

Input: Target Graph (M), number of expansion steps (k), minimum occurrence frequency f_{min} and a selected pattern matching algorithm $\lambda =$ alg-PM
Output: *score*, *FreqM*

For each vertex u in $V(M_t)$ **do**
  $P_{pivot(u,1)} \leftarrow u$
  $score_{(u,1)} \leftarrow$alg-PM$(M, P_{pivot(u,1)})$
  $f_{(u,1)} \leftarrow$ countFrequency$(score(u,1), P_{pivot(u,1)})$
  If $f_{(u,1)} > f_{min}$ **do**
    If $k >= 1$ **do**
      For $j : 2 \rightarrow k$
        $P_{pivot(u,j)} \leftarrow expand(P_{pivot(u,j-1)})$
        $score_{(u,j)} \leftarrow$alg-PM$(M, P_{pivot(u,j)})$
        $f_{(u,j)} \leftarrow$ countFrequency$(score(u,j), P_{pivot(u,j)})$
      end for
    end if
  Else printf(insufficient frequency of temporal match centred on u)
  end if
end for

Table 6.2: countFrequency function for counting the frequency of a subgraph P in M.

countFrequency

Input: *score* resulting from matching subgraph P_t in M_t, subgraph P_t, approximate match ratio (Rt)
Output: f

For each index i in *score* **do**
  If $score(i)/|V(P_t)| >= Rt$ **then**
    $cnt \leftarrow cnt + 1$
  end if
end for
$f \leftarrow cnt/|V(P_t)|$

Pseudo-code of λ-PD. The λ-PD algorithm takes as inputs the recorded graph M_t of a target process model where frequent patterns would be searched for, a parameter k defining the maximum number of expansion steps for initial temporal matches centred in each vertex of M_t, a parameter f_{min} indicating the minimum occurrence frequency that a subgraph must have to be considered a frequent pattern and a selected pattern matching algorithm λ = alg-PM - presented in the previous chapter - that is used to identify occurrences of subgraphs in M_t. The outputs of the algorithm are two matrices (*score* and *FreqM*). *score* is a $m \times k$ matrix of vectors. Each entry (i, j) in *score* is a vector of length m, where m is the number of vertices in the recorded graph of M_t. The vector contains the results of matching a subgraph $P_{pivot(i,j)}$ originated in the vertex i and created by means of j expansion steps that included its neighbours. k is the maximum number of expansion steps considered by the algorithm. *FreqM* is a $m \times k$ matrix whose entries indicate the frequency of subgraphs representing potential frequent patterns. These subgraphs are created from expansion steps indicated in the Line 8 of Table 6.1. Subgraphs in each step continue their expansion if their frequency in M_t remains greater than f_{min} and performed steps are less than k. Occurrence frequency of a subgraph is calculated with the function countFrequency in Table 6.2. For countFrequency, the ratio between the result of matching a subgraph (pattern) $P_t \subset M_t$ and the number of vertices in P_t is compared to an approximate match ratio Rt. If $Rt = 1$, the match between a non-identical occurrence of P_t and P_t is exact and it does not consider surjection. If $Rt > 1$ the counted occurrence involves an overlap. If $Rt < 1$, it indicates a partial match - for details of exact, partial and overlapped pattern instances, please refer to Section 4.4. The last column in *FreqM* indicates the frequency of subgraphs of M_t originated from each vertex in $V(M_t)$ and expanded k-times with its neighbours.

6.3 Summary

The contribution of this chapter is to relate the well-known problem of frequent subgraph discovery to the finding of potential services based on frequent process substructures. This idea promotes process-centric service reuse and can provide a mechanism to discern redundant software components supporting similar or equivalent process steps in complex and often de-integrated processes across organisations.

A solution for the frequent subgraph discovery problem is proposed. The proposal relies on the pattern matching solution provided in previous chapters and focuses on a single-graph scenario. It processes graphs one at a time and the solution provided is restricted to count overlapped subgraphs (patterns) as a single occurrence – i.e., it identifies overlapped instances but it can not differentiate them. Other proposals in the literature can be more adequate in situations where overlaps

are common or parallel processing is required. However, the inherent complexity of process descriptions, which can involve an elaborated typing and multiple attribute values in graphs – including descriptions in natural language – certainly defines a more complex scenario compared to more simple graphs considered in the literature. The complexity of the frequent pattern discovery problem in graph-based process descriptions goes beyond the structural problem addressed when identifying frequent subgraphs, it also includes matching of complex types and attributes describing process elements.

Before explaining in detail the results of an evaluation for the matching and discovery techniques (Chapters 5 and 6) proposed in the context of the LABAS framework for process and application integration (Chapter 3), a reference to the general limitations of the pattern discovery technique is provided. These indicate that the proposed technique may perform poorly when the occurrences of the frequent graph are highly overlapped. Also, as the pattern *discovery* technique is based on a pattern *matching* technique, the latter could present limitations during matching activities when dealing with unlabeled and highly connected graphs [Messmer 2000]. To the best of the author's knowledge there are not empirical studies indicating if it is frequent to find real process descriptions which have highly overlapped frequent pattern instances. Intuitively, if frequent subgraphs (patterns) are used to guide the design of reusable services, they should not overlap, since another of the important design guidelines for services is that service implementations should be – ideally – decoupled from each other. Thus, highly overlapped instances can be discarded from the results since they are not helpful for end users. Unlabeled and highly connected graphs are not representative of process graphs. Process graphs often have labels that identify the different process elements [Aguilar-Saven 2004] and connections between process elements are frequently upper-bounded to six or eight out/in-going edges per process element [Golani 2003], which can not be considered a property of a highly connected graph for the case of realistic medium size process graphs whose size is in the order of hundred of vertices.

A comparative study covering other techniques for frequent pattern discovery based on different paradigms, for example the approximate solutions in [Kuramochi 2005] or cluster-based approaches such as in [Jung 2006] are required to recommend an adequate solution to different process scenarios, specially in the presence of overlapped frequent patterns. A number of proposals for sequential and parallel calculation of the frequent subgraph mining problem to discover interesting patterns has risen the attention in diverse applications scenarios such as analysis in social networks, molecular compounds and document-based information retrieval [Han 2007], [Wang 1995], [Kuramochi 2005], [Bringmann 2008]. Solutions in those scenarios seem to require major efforts to be translated to the process and work-

flow settings [Greco 2005]; hence, dedicated solutions as the one presented here are required.

Given the limitations to distinguish non-identical overlapped frequent subgraphs in this particular proposal for discovering frequent patterns on graphs, investigations on how other graph-based techniques proposed in the literature can be adopted in the single-graph setting scenario for large and complex process models is defined as a line of future work. Another envisioned and promising line of future work refers to the use of spreading activation mechanisms [Crestani 1997], [Cohen 1987], [Faloutsos 1995] over dynamic and weighted graphs for frequent pattern discovery on run-time processes. Possible future work at the end of the thesis discusses this idea in more detail (see Section 10.3.2).

Chapter 7

Evaluation of LABAS Framework

Contents

7.1 Overview

Previous chapters have presented a pattern-based framework and techniques to support the design of enterprise processes and applications integration solutions based on services. Services are the intermediary architecture elements relating the business operation to the process-wide application architecture elements. This chapter describes how the proposed framework is evaluated in regard with the thesis objectives and it focus is on the overall framework and its support to architecture modelling activities. Tool support for the framework is also described. The next chapter focuses on the evaluation of the proposed pattern matching and discovery techniques.

7.1.1 Influenced System Quality Characteristics

During the design of service-based systems to integrate enterprise processes and applications there are a number of challenges that influence the quality characteristics

of the final architectural solutions. One part of the evaluation of this work refers to quality characteristics that can benefit from the proposed pattern-based framework.

Nowadays enterprise software systems no longer evolve as separate entities but evolve integrated with each other in a complex interrelated system [Land 2003]. Changes are constant and process and application integration systems have to be continually re-designed to meet new requirements. Maintainability is a central characteristic in this scenario and it helps the organisations' capacity to change. For process and application integration systems, the analysis and design stages are predominant in terms of cost and consumed time [Hohpe 2004].

On the other hand, functional characteristics are critical during analysis and design stages. Functional suitability during initial stages of development is a basic and important characteristic in regard with requirements coverage. Also, compliance to process regulations is a core aspect of process integration systems [Daniel 2009], [Kharbili 2008]. Identifying early non-compliance to process regulations can reveal future failures in the implementation of a system [Lu 2008].

Considering the categorisation for quality characteristics in the ISO/IEC 9126 standard and its corresponding update in the ISO/IEC 25000 standard series, the explored quality sub-characteristics in this investigation are derived from *functionality* and *maintainability* characteristics. These are *functional suitability*, *functional compliance*, *changeability*, *analysability*, *reusability* and *traceability* (the latter is discussed in the context of *analysability*). Appendix B provides descriptions of these quality sub-characteristics in relation to requirement analysis and architecture design stages[1] in the development of process and application integration systems.

7.1.2 Evaluation Strategy

The overall evaluation strategy adopted in this work involves three different approaches,

- *Experiments.* The core technical contribution of this work are the pattern matching and discovery techniques used within the context of the proposed LABAS framework. The framework uses patterns to support the design of services and architectures. In order to evaluate the techniques, an empirical evaluation involving a set of experiments is adopted to explore the effectiveness and efficiency of the proposed algorithms for pattern matching and discovery. Chapter 8 describes the experimental setup and results.

[1] According to ISO/IEC 25010 std., at the earliest stages of development, only resources and processes can be measured. When intermediate products (specifications, source code, etc.) become available, these can be evaluated by the levels of the chosen internal measures. These measures can be used to predict values of the external measures.

- *Case study & ALMA.* An scenario based method is used with two case studies to assess the proposed pattern-based framework with regard to its benefits to maintainability. The ALMA method [Bengtsson 2004] was selected among other scenario based methods such as ATAM [Kazman 2000] because it focuses on maintainability and it can be used at various stages of development. This work is centred in the initial stages of development, when domain models are analysed and services are designed. The case studies and scenarios used with the ALMA method provide rich examples for analysing the sub-quality characteristics of interest in regard with the use of patterns and modelling support in the context of the proposed architectural framework. Details on the ALMA method and its use with the case studies are provided in Section 7.2[2].

- *Interviews.* A complementary assessment based on interviews supports the evaluation process horizontally. Based on the interviews, the view of a number of practitioners from industry regarding model-based analysis and design techniques is explored. Chapter 9 describes an overview of the interviewing process and a discussion of the results.

Expected benefits for end users and organisations. Beneficiaries of an organised framework and pattern-based techniques to address process and application integration are those end users whose roles are business analysts and IT architects, possibly broadening the range to other related roles such as enterprise architects and solution architects [Tandon 2007]. Inexperienced end users could benefit from the proposed tools providing a medium to reuse design knowledge in the form of patterns. Pattern matching and discovery could facilitate the identification of similar (and possibly redundant) designs. As a consequence of identifying similar designs, experienced end users could consider the use of automated tools to facilitate the benchmarking of their solutions. At an organisational level, benefits of using an organised architectural approach (framework) and design reuse support (pattern-based techniques) are expected from reducing the complexity of integration solutions and its associated costs regarding maintainability. Inefficiencies such as unnecessary use of time from domain and process experts due to lack of analysis tools, re-working of service designs due to redundances and misalignment between process and software levels are issues that could be addressed with the help of the proposed approach.

[2]Note that in order to follow the same sequence of how the framework (first) and the techniques (later) were introduced in previous chapters, the ALMA method and its use to evaluate the LABAS framework are presented first - in this chapter - and the experimental evaluation of the central contribution (pattern matching and discovery techniques) is presented later in Chapter 8. This ordering represents how this work went from general and contextual aspects to detailed and central aspects of the contribution.

7.1.3 Specific Challenges, Solutions and Evaluation Methods

This section refers to the addressed challenges (problems), proposed solutions and adopted evaluation methods. Table 7.1 provides a summary that includes references to (sub-)quality characteristics affected by these problems. The targeted quality characteristics are those early mentioned in Chapter 1 (hypothesis), i.e., maintainability, functional suitability, functional compliance and traceability. Throughout the chapter, instead maintainability, derived sub-characteristics are analysed: changeability, analysability and reusability. Traceability is discussed in the context of analysability.

Table 7.1: Summary of problems, proposed solutions and evaluation methods

Challenge	**QSC (*)**	**Solution (LABAS)**	**Evaluation method**		
			ALMA	Experiments	Interviews
P1	Suitability, Analysability Changeability	Framework	✓	-	✓
P2	Analysability Changeability	Traceability support	✓	-	✓
P3	Reusability	Pattern support	✓	✓	✓
P4	Compliance	Pattern support	✓	✓	✓
P5	Reusability Compliance	Pattern support	-	✓	✓

(*) : Affected quality sub-characteristics.
P1 : Separate models for process, domain and architecture descriptions.
P2 : Misalignment between process and architecture levels after changes.
P3 : Ineffective or inefficient use of design knowledge.
P4 : Lack of automated support for checking regulatory process compliance.
P5 : Undesired time consumption and errors due to manual efforts during pattern identification.

P1 : Separate models for process, domain and architecture descriptions

- *Problem description.* According to [Kazman 2000] architectures have to be well documented, it includes static and dynamic views and the use of an agreed-on notation that all stakeholders can understand with a minimum of effort. These requirements are difficult to achieve if the architecture spans several processes across an organisation(s) and involve different applications with heterogenous architectures. This situation can be frequent for process and application integration scenarios [Linthicum 2000], [Johannesson 2001]. A basic requirement in this scenario is to have an integrated view of the business operation (processes) and its supporting application architecture. Maintaining different non-connected models for all elements involved in an integration problem puts in risk the adequate satisfaction of integration needs and correctness of implementation [Johannesson 2001], thus affecting functional *suitability*. Also, more

time is required to analyse what are the consequences over other elements in different layers, thus influencing *maintainability*.

- *Proposed solution.* A modelling framework organised in a layered architecture is presented in Chapter 3. The layered architecture includes three main layers encompassing process and domain models, and application and service architectures.
- *Affected (sub)-characteristics.* Functional suitability and maintainability (analysability and changeability).
- *Evaluation method.* The Architecture Level Modifiability Analysis (ALMA) method [Bengtsson 2004] is considered to evaluate changeability and analysability sub-characteristics derived from maintainability. Suitability is demonstrated in a case study. The ALMA method and case study are presented in Section 7.2.1.

P2 : Misalignment between process and architecture levels after changes

- *Problem description.* The problem of misalignment between business and software levels has been investigated since decades and from different perspectives [Luftman 1999], [Henderson 1993]. From the modelling perspective, a fundamental aspect is the capacity of interrelating the different models involved in both, business and software levels [Lankhorst 2005]. When changes at any of the levels occur, their impact on other layers has to be assessed. Problems associated to unknown effects can result in high costs for re-design and implementation [Chen 2005].
- *Proposed solution.* Explicit modelling of dependencies and active change management across layers of LABAS framework (Chapter 3).
- *Affected (sub)-characteristics.* Maintainability (analysability and changeability).
- *Evaluation method.* Architecture Level Modifiability Analysis (ALMA) method [Bengtsson 2004].

P3 : Ineffective or inefficient use of design knowledge

- *Problem description.* Each time an integration project takes place, analysis and design activities initiate the development. Understanding the enterprise context and the problem domain associated to the integration project is the most complex and time consuming part of the process [Linthicum 2000]. Reusing proven design solutions like patterns has been widely promoted as a medium to reduce costs and development time while benefiting design quality – see e.g., [Gamma 1993], [Buschmann 2007], [Barros 2007], [Zdun 2007b]. An inefficient use of design knowledge in the form of patterns could reduce the potential cost

savings of using proven designs, including savings due to reduced efforts of development and testing [Vokac 2004], [Buschmann 2007]. If the conceptualisation and infrastructure to create, manage and reuse patterns does not exist, the previously mentioned benefits are more difficult to achieve. Automated support for architectural design has a number of contributions for at least two decades [Shaw 1996a]. With the emergence of new architectural approaches such as service-based architectures and the increase of abstraction levels moving traditional software integration support (EAI tools) closer to business levels (BPM tools) [Hill 2009], [Oracle 2008], automated support becomes more specialised and sophisticated, including identification of process artifacts to facilitate reuse and change management.

- *Proposed solution.* A framework for enterprise process and application integration that allows management of design knowledge in the form of patterns. Patterns can be at process model, and service and application architecture levels. A family of techniques is proposed to match and discover patterns in graph-based representations.
- *Affected (sub)-characteristics.* Maintainability (reusability).
- *Evaluation method.* A case-study based approach is adopted. It uses the Architecture Level Modifiability Analysis (ALMA) method [Bengtsson 2004] with scenarios that demonstrate differences in reusability. Techniques providing automation to match and discover patterns are assessed through an empirical evaluation. Description and results are provided in Section 8.

P4 : Lack of automated support for checking regulatory process compliance

- *Problem description.* Business process compliance has become a relevant concern for organisations since legislative and regulatory environments have increasingly been introduced [Ghose 2008]. Process and application integration projects cover both process and the underlying software layers. Ensuring compliance at business process level is a crucial feature for process-centric integration systems. Compliance has traditionally been achieved through efforts performed by a auditing experts with poor or nonexistent automated assistance [Kharbili 2008]. Manual efforts for checking compliance and lack of techniques and tools supporting this task may introduce errors and significant time regarding regulatory process compliance. Errors in turn, can introduce weaknesses that can result in fraud, organisational misconduct and loss of organisational reputation [Daniel 2009].
- *Proposed solution.* As one possible alternative to improve the lack of automated support for checking regulatory process compliance, the family of pattern

matching techniques presented in Chapter 5 is used to check if process patterns defining regulations over the operation of businesses are completely or partially instantiated, or if they are not presented at all.

- *Affected (sub)-characteristics.* Functional compliance.
- *Evaluation method.* Using the case study presented in Section 7.2, process compliance is discussed in relation to hypothetical regulations formulated as process patterns. Techniques providing automation to match and discover process patterns are assessed through an empirical evaluation. Description and results are provided in Chapter 8.

P5 : Undesired time consumption and errors due to manual efforts during pattern identification

- *Problem description.* As mentioned previously, increased time consumption and susceptibility to errors due to services design can occur due to manual efforts during the analysis of dependencies in elements from different models associated to the integration problem, identification of services and process abstractions (patterns) or when checking compliance to process regulations. Automated support for architectural design has a number of contributions for at least two decades [Shaw 1996a]. With the rising of new architectural approaches such as service-based architectures and the increase of abstraction levels moving traditional software integration support (EAI tools) closer to business levels (BPM tools) [Hill 2009], [Oracle 2008], automated support becomes more specialised and sophisticated, including identification of process artifacts to facilitate reuse and compliance management.
- *Proposed solution.* A family of pattern matching and discovery techniques to assist the definition of new services and compliance to process regulations is presented in Chapters 5 and 6.
- *Affected (sub)-characteristics.* Maintainability (reusability) and functional compliance.
- *Evaluation method.* Techniques providing automation to match and discover patterns are assessed through an empirical evaluation. Description and results are provided in Chapter 8. References to the use of these techniques in scenarios used with the ALMA are also provided.

7.2 ALMA-based Analysis of Case Studies

The ALMA method is used with two case studies that include a number of different scenarios that provide rich examples for analysing changeability, analysability, reusability, traceability, functional compliance and suitability with regard to the use of patterns and modelling support in the context of the LABAS framework. The case studies and scenarios capture models, model changes and abstractions in the form of patterns in different layers of the integration problem (process model, service architecture and application architecture layers). The case studies capture a scenario of integrated financial network services from an application perspective and a scenario of a typical process in the e-commerce domain where customers and businesses interact. Note that the cases have limitations with regard to the need of an empirical justification where scenarios are confirmed by a relevant number of analysts or architects and concrete implementations can be controlled. Beyond these limitations, the aim of considering these scenarios has been to represent common situations of processes and applications integration problems in organisations.

7.2.1 Architecture-level Modifiability Analysis Method

ALMA method is a scenario-based method designed for predicting maintenance efforts, assessing risk and comparing different candidate architectures [Bengtsson 2004]. The ALMA method consists of five steps[3], described as follows.

- *Set goal*: setting the analysis goal;
- *Architecture description*: giving a description of the relevant parts of the software architecture and their configuration;
- *Elicit scenarios*: finding the set of relevant change scenarios;
- *Evaluate scenarios*: determining the effects of the set of scenarios; and
- *Interpret results*: drawing conclusions from the analysis results.

Even though the main target of the ALMA method is to assess modifiability of a system, the method is adopted to analyse a closely related quality characteristic: changeability. Most of the aspects covered by the definition of changeability in Section B.4.1 overlap with the definition of modifiability in ALMA, that is defined as the ease with which a software system can be modified to changes in the environment, requirements or functional specifications.

On the other hand, *analysability*, as a characteristic that benefits the easiness to modify a software system, can also use the change scenarios derived in ALMA. Moreover, these scenarios provide rich examples to assess functional *suitability* and

[3]Note that while performing the analysis, sequentiality of steps is not strict and it is often necessary to iterate over various steps.

compliance. The case studies presented in the next sections use the steps defined in the ALMA method to assess the benefits of the proposed framework and techniques to changeability, analysability, suitability and functional compliance.

Assumptions in this work. A number of assumptions are considered when applying ALMA to the case studies in the next sections. These involve the existence of models describing the different layers of the integration problem, that all models can be translated to a single graph-based notation, and the existence of a mechanism to access models even though they could be located remotely in a distributed environment.

Note that process models may have been created manually (during traditional documentation activities) or automatically by using tools to mine process logs, e.g., [Aalst 2007], [Bae 2006], [Greco 2005]. The only conditions for these models are the last two listed above. Also, note that the previous assumptions are valid for future case studies involved in further research for assessing the proposed framework and techniques, which can be applied in early stages of analysis and design for service-based process and application integration systems development.

7.2.2 Loan Management (LM) Case

7.2.2.1 Analysis goals

The analysis goal is to get a comparative analysis of the costs of modifying the LM process and its supporting systems. The emphasis is on analysing the relative benefits for the proposed framework and a manual-based approach regarding analysability, changeability and functional compliance support. The analysis is a post-mortem analysis and therefore there is no need to normalise the weights of each change scenario in ALMA [Bengtsson 2004]. The change scenarios involve modifications resulting from the improvement of a bottleneck activity that affected the operation of the business, the incorporation of new process regulations and technological updates. Changes resulted in the re-design of the *Loan Management* process and its underlying software.

7.2.2.2 Processes and software architecture

This case study looks at a scenario of integrated financial network services from an application perspective. The main process involved is a traditional loan management process. This is a basic banking operation and it represents a good example of a process in the financial domain. The aim in this case is to integrate the process's supporting software to facilitate the tasks performed by bank agents, specially those

considered to be a bottleneck. Also, internal rules for complying with loan management regulations would require modifications of some of the activities in the *Loan Management* process.

Figure 7.1 shows three business actors and their participation in a simplified loan management process. The process starts when a client requests a loan by email. A bank agent from a call centre calls the client to present him an offer. To provide the offer the agent needs to obtain client data, calculate the amount of loan offer and to call the client to explain the conditions of the loan (top of Figure 7.1). In subsequent steps of the process, the agent registers relevant information regarding the client's acceptance respect to the offer. If the client accepts the offer and the amount of money involved in the loan is sufficient to meet the sales goals of the bank, then a business alert is triggered. Due to this alert, a direct sales agent located near the client would visit her. The agent visits the client with the objective to establish a contract and obtain a signed contract document. After the contract is established and the agent is in the bank location, she registers the visit with her signature and reports the visit to her supervisor.

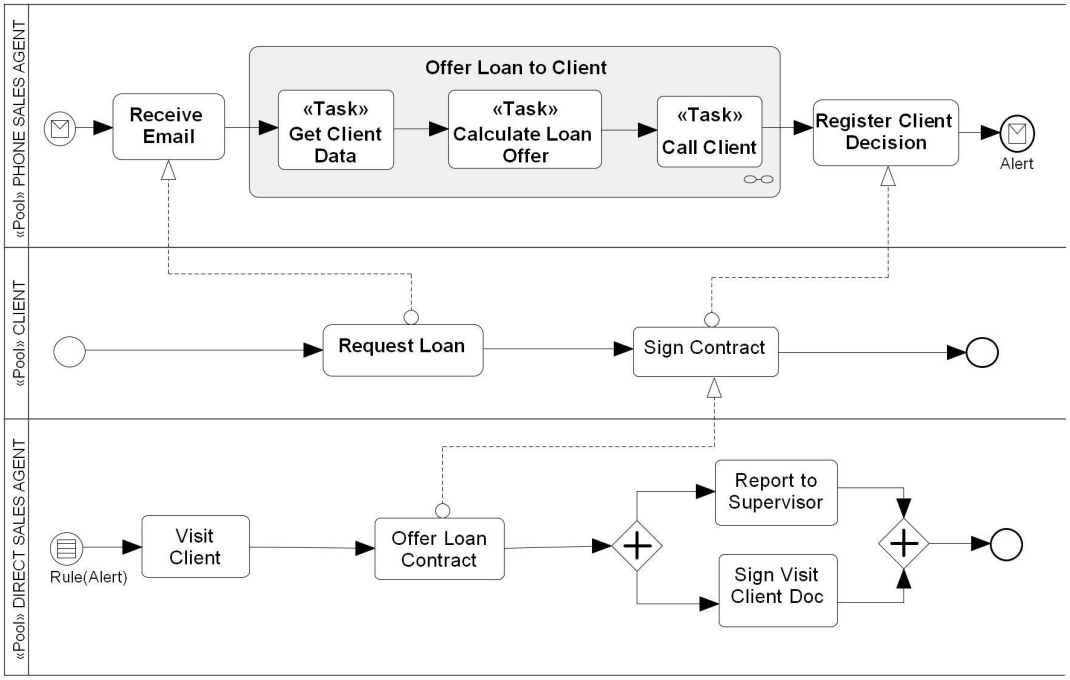


Figure 7.1: The *Loan Management* Process.

The applications supporting the process on the phone sales agent side and an extract of the main domain model elements are indicated in Figure 7.2 and 7.3, respectively. The involved applications are a mail server application (EMAIL-MNG), a customer relation management application (CRM), an application managing the

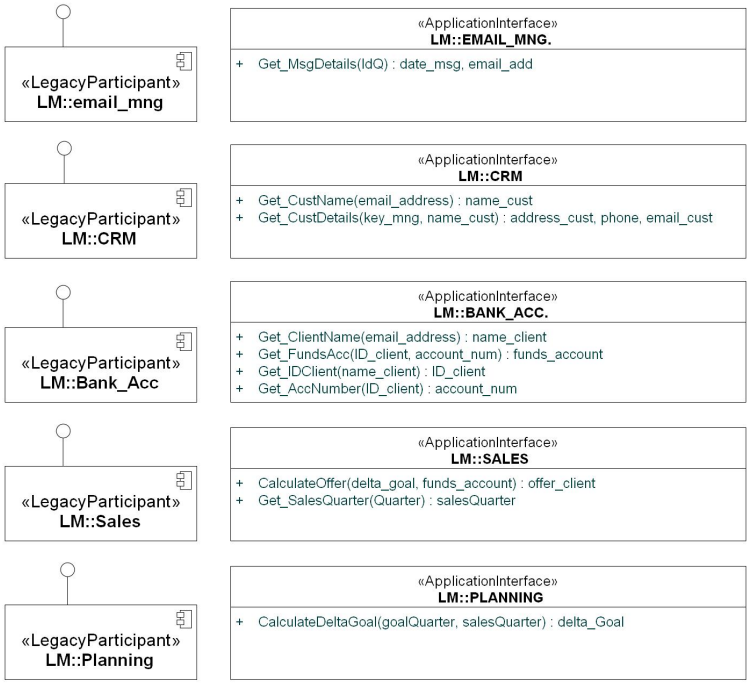


Figure 7.2: Applications supporting the LM process (phone sales agent role).

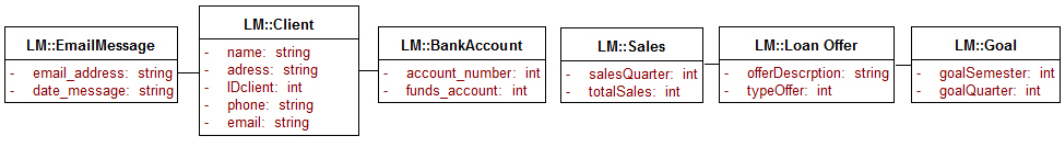


Figure 7.3: Relevant domain model elements, LM process - phone sales agent role.

client's bank account (BANK-ACC), and two applications managing the information of sales and plans of the bank, SALES and PLANNING, respectively. Available operations are shown in the application interfaces at the right side of Figure 7.2. The applications are isolated and the data manipulated during the process is managed by the phone sales agent. Other activities in the LM process have no direct software support.

7.2.2.3 Change scenarios elicitation

Two business scenarios and five technical scenarios describe requirements that are the basis of the change scenario elicitation. These scenarios involve changes at the process level triggered by requirements to comply with process regulations and new technological acquisitions. Also, the architecture levels are affected by new requirements for automating processes. At a software architecture level, the software components (existing and new ones) are iteratively configured according to patterns targeting different requests to improve quality characteristics.

First business scenario (B1). Managers at the bank asked IT architects if they could facilitate the tasks of the *phone sales agent* in order to reduce the time to attend the clients. They suggested to integrate the software supporting the agents work. Special attention to the *Calculate Loan Offer* activity is required, since it is one of the potential bottleneck activities in the process.

Second business scenario (B2). The internal rules to comply with loan management regulations define that after *direct sales agents* visit their clients, they must report these visits to their supervisors and sign a document registering them. After implementing mobile access to banking application functionalities, the previous process on the *direct sales agent* side is modified to the process illustrated in Figure 7.4. In the new process version, the agent can remotely report to the supervisor using a mobile device. In that way, she/he is only required to sign a physical document at the office.

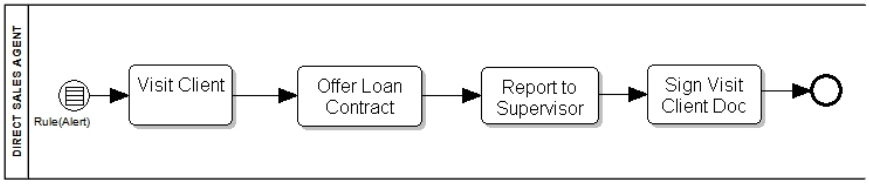


Figure 7.4: *Loan Management* process variation.

First technical scenario (T1=B1). In order to respond to the first business scenario and given the criticality of the *Calculate Loan Offer* activity, the first target applications to be integrated are those allowing to calculate the loan offer, these are the SALES and PLANNING applications. IT architects of the bank discussed the *Remote Procedure Invocation* pattern (RPI-PAT) [Hohpe 2004] as a well-known strategy to communicate between two different applications. They applied the pattern and a direct communication channel between SALES and PLANNING was created. Figure 7.5.a illustrates this situation.

Second technical scenario (T2). As a result of the changes in T1, agents could calculate loan offers easier and quicker. Managers at the bank asked the IT architects if they could do the same for all the applications used by the phone sales agents. The dedicated connection created between the SALES and PLANNING applications was a non-standard communication solution that tightly coupled the two applications. If the same strategy were used with the rest of the applications, the amount of dedicated connections complexity could increase to an upper limit of $n(n-1)$, where n is the number of involved applications. Maintainability of the overall solution could be degraded. IT architects decided to tackle the problem, but trying to avoid point-to-point connections whose updates could significantly increase costs of maintenance due to the constant application upgrades that were taking place. The new integration strategy would try decouple applications while providing some communication solution. They looked at the *Service Messaging* (SM-PAT) and *Legacy Wrapper* (LW-PAT) patterns [Erl 2008] as design solutions to enable messaging oriented communication and to benefit from loose coupling between application functionality [4].

The new architecture design involved the creation of a messaging communication schema between applications and the addition of a new central component (CMM) to persist and manage messages. The intention of the IT architects was to provide a more reliable communication, independent from particular applications. Loose coupling was addressed by wrapping the legacy applications and offering their relevant functionality as services. The interaction between services was coordinated through messages managed in the CMM component.

Additionally, given the high level of operation of the three sequential tasks involved with *Offer Loan to Client* activity, a dedicated (composed) service was designed to serve the composed activity. Figure 7.5.b illustrates the main elements of the new architecture.

[4] SM-PAT and LW-PAT patterns can be seen as predecessors of the *Messaging* pattern in [Hohpe 2004] and the *Adapter* pattern in [Gamma 1995], respectively.

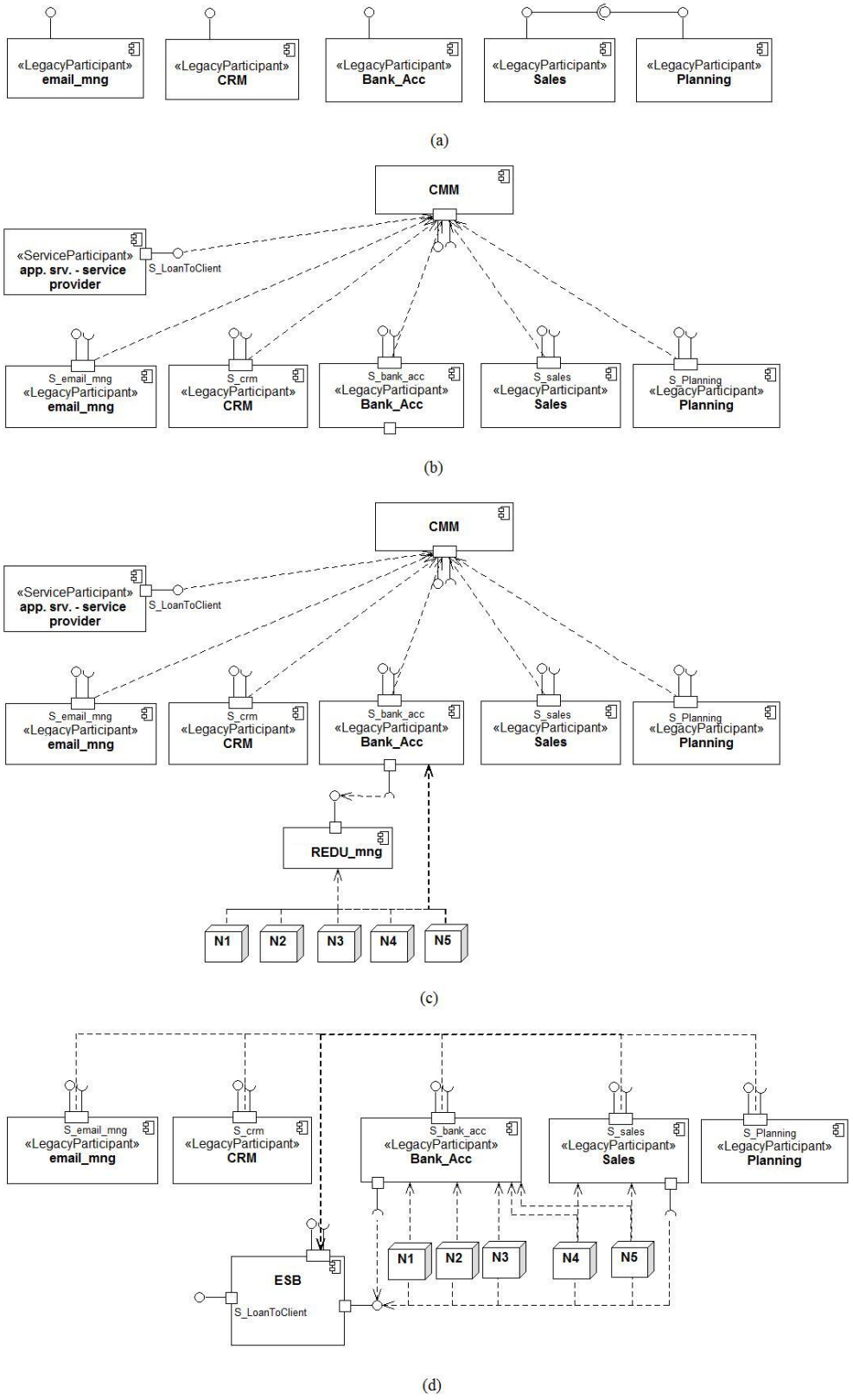


Figure 7.5: Architecture model evolution.

Third technical scenario (T3). After adopting a service-centric strategy to integrate applications and support processes, a new requirement regarding availability of services imposed an additional change on the service architecture. The service exposing functionality from the BANK-ACC application was reused in several other activities and the increasing demand on it caused an unsatisfactory performance level. Moreover, managers indicated to the IT architects that there were plans to expand the organisation that would create a growing demand for the already over-requested service.

The IT architects discussed the strategy of the *Redundant implementation* (RI-PAT) pattern [Hohpe 2004] and decided to apply the pattern for the service exposed by the BANK-ACC application (S-BANK-ACC) with the aim to improve availability of the service. The service would have a redundant implementation with five infrastructure components and one redundant infrastructure manager (REDU-MNG) component supporting it. The resultant architecture is illustrated in Figure 7.5.c. A negative consequence of applying RI-PAT was that REDU-MNG became a unique point of failure.

Fourth technical scenario (T4). Over time, the IT architects noticed a decreasing demand on the service exposed by the S-BANK-ACC (S-BANK-ACC), and increasing demand on the service exposed by the SALES application (S-SALES). Due to the unsatisfactory performance of the S-SALES service, the managers asked IT architects to solve the problem, but to keep maintenance and operational costs as low as possible.

The IT architects decided to redistribute the five nodes implementing the redundant implementation for S-BANK-ACC and to apply the RI-PAT pattern on the S-SALES service. The final configuration left three nodes for the S-BANK-ACC and two nodes for S-SALES as shown in Figure 7.5.d. On the other hand, the CMM, Service Provider and REDU-MNG components from Figure 7.5.c were replaced by a new infrastructure component, an Enterprise Service Bus (ESB). The ESB would jointly manages the redundant components, messaging tasks and service exposition. They aimed at lowering the operational costs of maintaining the two architecture components to manage messages separated from the access to services. There was still a single point of failure, in this case the ESB.

Fifth technical scenario (T5=B2). In response to the second business scenario, mobile access to the S-BANK-ACC is provided to *direct sales agents*. The responsibility of proving mobile support was directed to the ESB component. There were no chances to models in Figure 7.5.d. However, compliance to regulations required a revision at business level.

7.2.2.4 Scenario Evaluation

After scenarios and related architectures have been described, the modification efforts from one scenario to the other are analysed. The analysis focuses on modifiability and analysability regarding traceability, pattern coverage-based complexity and functional compliance. See Appendix B for a description of quality characteristics and associated metrics. Table 7.2 summarises the scenario evaluation. The rest of the section describes how these results were obtained and it discusses the advantages of the proposed framework and techniques in comparison to an unsupported (hand-crafted) and more human-centric approach.

Table 7.2: Scenarios evaluation - LM case study

	LABAS					MANUAL				
	B1=T1	T2	T3	T4	T5=B2	B1=T1	T2	T3	T4	T5=B2
Traceability	1	1	1	1	1	0.4	0.7	0.8	0.4	0.4
Complexity										
Q1 (pattern coverage)	+	+	+	+	+	±	±	±	±	±
Q2 (pattern coverage)	+	+	+	+	+	±	±	±	±	±
Q3 (affected properties)	±	±	±	±	±	±	±	±	±	±
Changeability										
Q4 (changeability)	+	+	+	+	+	−	−	−	−	−
Q5 (modifiability)	±	±	±	±	±	−	−	−	−	−
Q6 (scalability)	±	±	±	±	±	−	−	−	−	−
Functional compliance										
Q7 (documentation)	+	+	+	+	+	+	+	+	+	+
Q8 (automation)	+	+	+	+	+	−	−	−	−	−
Q9 (quantification)	+	+	+	+	+	±	±	±	±	±

+, ± and − indicate direct support, indirect support and non-support, respectively.

Q1: What patterns are instantiated in architecture and process models?
Q2: What pattern roles fulfill determined elements in models?
Q3: What properties are affected by a change in a pattern instance?
Q4: Can individual costs of adding/eliminating components be registered and maintained?
Q5: Can costs of modifiable transition mechanisms be obtained automatically?
Q6: Can costs of scalable transition mechanisms be obtained automatically?
Q7: Can process regulations be documented?
Q8: Can compliance pattern instances be identified in actual processes automatically?
Q9: Can be compliance pattern support be quantified?

Traceability. Traceability is a characteristic that can indicate the effectiveness of documentation and design structure mapping functions from requirements to implementation (see Appendix B). Enterprise process and integration systems are complex systems that involve a large number of interrelated elements in different layers. The risk of poor requirements satisfaction due to insufficient or incorrect analysis in large system is likely when there is no awareness of the impact of changes in related elements from other layers [Lankhorst 2005], [Land 2007].

A traceability model relating elements of the integration problems in different

layers can be obtained by following the suggested steps in the LABAS framework to create models in the BAIL layer (see Section 3.2.1 for details). The model in Figure 7.6 illustrates trace links between process steps (top), domain model elements (middle) and applications (bottom) involved in the integration problem of scenario T1. The illustrated models focus on the *phone sales agent* role.

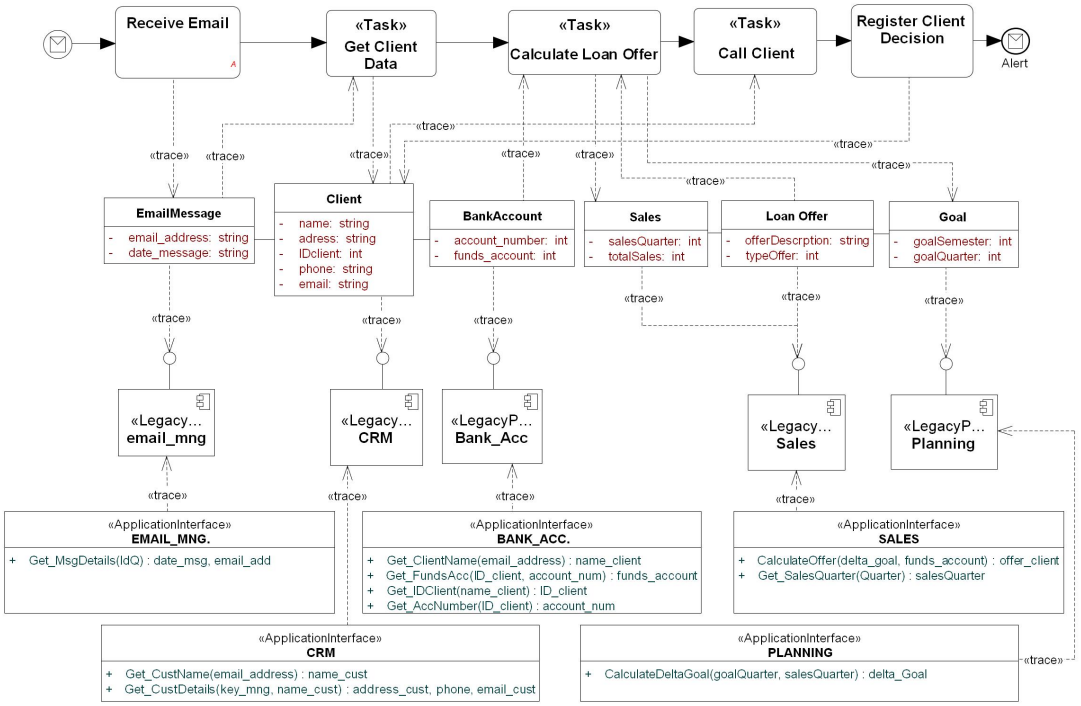


Figure 7.6: The *Loan Management* Process for the *Phone Sales Agent* role.

According to Equation B.4 (Appendix B), traceability would equal one for all scenarios T1 to T4. This considers the ratio between the number of effectively traced elements ($\sum_i ETE_i$) and the total number of traceable elements for such a model ($\sum_i TE_i$), excluding trace links between applications and their interfaces. Table C.1 in Appendix C details the model-to-model trace links for model elements in different layers for scenarios T1 to T4 and *phone sale agent* role. Newly added elements in each scenario are associated to new trace links. For T5, there are no changes in models or new created trace links.

An approach that does not consider any kind of traceability support can make the analysis of a system a very complex task and can consequently negatively affect its analysability characteristics and therefore, maintainability. Without traceability support, any possible relation between model elements has to be manually analysed. During modifications, this can increase the costs of maintenance. This can happen to the models in Figures 7.1, 7.2, 7.3, which are the only available documentation

during modifications of the supporting LM process architecture.

If an unsupported/hand-crafted approach approach is assumed for the LM case, trace links would be established each time a new modification during scenarios T1 to T4 is required. Table 7.2 indicates the variations for estimated values of traceability in scenarios T1 to T4. These values assume that new trace links are manually established after analysing models in each scenario.

- At the beginning of T1, there are no established trace links between model elements. Traceability is initially considered equal to 0, where $[\sum_i ETE_i = 0]/[\sum_i TE_i = 5+6+5]$. New trace links are created after analysing the *Calculate Loan Offer* activity. Trace links are established between the activity and the *BankAccount*, *Sales* and *Loan Offer* domain model elements. In turn, these domain model elements are traced to the SALES and PLANNING applications. Traceability is then increased from zero to $[1+3+3]/[5+6+5] \approx 0.4$.
- For T2, new relations associated to the CMM component and services S-EMAIL-MNG, S-CRM, S-BANK-ACC, S-SALES, S-PLANNING and S-LOAN-CLIENT have to be created. Traceability would change from 0.4 to $[5+6+5]/[5+6+6+6] \approx 0.7$.
- Similarly, traceability in T3, where the REDU-MNG and N1 to N5 components are added, would be $[5+6+6+6]/[5+6+6+7+5] \approx 0.8$.
- Finally, for T4, CMM and REDU-MNG are not longer part of the architecture and, therefore, all trace links from services to CMM should not exist. Also, N4 and N5 changed their relations from BANK-ACC to SALES application and their associated trace links should be updated. An estimated value for traceability would be $[5+6+0+0]/[5+6+6+6+5] \approx 0.4$.
- For T5, there are no changes in models. Hence, traceability is still 0.4.

Complexity. Software complexity refers to the characteristics of a software that makes it easier or more difficult to understand and maintain. As described in Section B.4.2, analysing architectural complexity in terms of pattern coverage can complement less reliable metrics based on coupling and cohesion or fan-in and fan-out [Kazman 1998]. Architecture complexity based on pattern coverage can be also extended to process-centric models [Gruhn 2006], [Mendling 2007], [Hirzalla 2009]. During architecture and process model modifications in change scenarios, some relevant questions related to pattern coverage are
Q1: What patterns are instantiated in architecture and process models?
Q2: What pattern roles fulfill determined elements in models?
Q3: What properties are affected by a change in a pattern instance?
Table 7.2 indicates how answers to these questions Q1 to Q3 could be supported by

the LABAS framework and an unsupported (hand-crafted) and more human-centric approach.

An unsupported (hand-crafted) and more human-centric approach does not rely on tools providing pattern manipulation support, instead answers to the questions above rely on the architects' knowledge and possible existing pattern documentation. A common problem is that over time, different people are analysing models in the different scenarios, and awareness that a pattern(s) has been instantiated in a previous scenario is very unlikely. This situation not only affects *analysability* when assessing complexity of a system based on its pattern coverage, but also decreases the possibility of reusing encapsulated design knowledge in the form of patterns [Land 2007].

The LABAS framework proposes a range of facilities for pattern support. It goes from pattern documentation, pattern identification to pattern-based model transformation and combination. This thesis in particular addresses automated pattern identification (Chapters 5 and 6). The support of the framework to the questions above considers the wide range of proposed facilities, going beyond those specifically implemented in this work and evaluated in the next chapter. The framework involves a more comprehensive perspective than the scope of the pattern identification techniques.

Throughout the LM case's change scenarios, several patterns were considered during architecture modifications. From one scenario to the other, questions such as Q1 to Q3 were asked. Questions Q1 and Q2 can be answered using the proposed pattern matching techniques. These help to identify patterns instances and their elements in models (Chapter 5). Question Q3 would require documentation efforts to associate architecture properties to patterns. This information can be maintained in pattern templates from a pattern catalogue in LABAS (see Section 3.2.2). Effects on instantiated patterns due to architecture modifications are of particular interest, especially if (possibly) negative effects on quality attributes have to be analysed [Harrison 2007], [Harrison 2008]. Table 7.3 provides an example of how relations between patterns and a number of architecture properties can be utilised to analyse the impact of changes on architectures. The example involves scenarios T1 to T4 and RPI-PAT, LW-PAT, SM-PAT, RM-PAT, RI-PAT and ESB-PAT patterns. The relations in Table 7.3 indicate if a pattern has a positive, negative or neutral effect on an architecture property.

Modifiability and Scalability (sub-characteristics of changeability). Changeability, as described in Section B.4.1 involves modifiability and scalability. Modifiability can be calculated as the number of possible paths allowing the transit from a determined design to possible designs that have an attribute added to or subtracted

Table 7.3: Relations between architecture properties and patterns.

Arch. property	RPI-PAT	LW-PAT	SM-PAT	RM-PAT	RI-PAT	ESB-PAT
ap1	↑	↑	↑	–	–	↑
ap2	↓	↑	–	–	–	–
ap3	↓	–	–	–	–	↑
ap4	–	–	↑	↑	–	↑
ap5	–	–	–	–	↑	–
ap6	–	–	–	–	↑	↓
ap7	↓	↑	–	–	↓	↑

↑ : positive contribution, ↓ : negative contribution, – : neutral

ap1: communication between applications
ap2: standardised access to applications
ap3: decoupling between applications
ap4: messaging reliability
ap5: performance – throughput of processed messages
ap6: availability of offered functionality
ap7: maintainability

from its attribute set, but whose transition mechanism cost less than an acceptability threshold. On the other hand, scalability refers to raising or lowering the value of an attribute from a architecture design whose transition mechanism cost less than an acceptability threshold. A basic requirements for a tool aiming to support architecture changeability calculation is that it should be able to answer questions such as
Q4: Can individual costs of adding or eliminating architecture components be registered and maintained?
Q5: Can costs of modifiable transition mechanisms be obtained automatically?
Q6: Can costs of scalable transition mechanisms be obtained automatically?

An unsupported (hand-crafted) and more human-centric approach to calculate the transition mechanism cost would include the costs of analysing, designing and implementing the modification. Assuming that implementation costs would be equal between a hand-crafted or LABAS-based approach, then costs of analysis and design would be the ones varying.

Consider the change scenarios T1 and T4. To calculate costs of integration, the first step is to identify what data is being processed in the *Calculate Loan Offer* activity, then to identify what applications store or process this data, and subsequently sketch an abstract architecture (early design solution) focused on the identified data flow. In the manual-based approach, there are no explicit relations between model elements. Also, costs estimation starts with the costs of individual modifications and then these are (manually) aggregated to cover them all and to obtain an integrated cost figure. Modelling support in LABAS for traces and model elements can contribute directly to Q4, and can increase automation during the aggregation activity referred to in Q5. Support for Q5 is not directly implemented in the proposal, and it

would require extensions to the modelling framework in Section 3.2.1.1. Extensions would consist of adding tagged values to model-to-model trace links and model elements in layers. For instance, in scenario T2, individual costs to encapsulate application functionality as services and the cost of establishing communication with the CMM component can be captured in tagged values. Subsequently, these values can be aggregated to obtain an integrated cost figure. In Table 7.2, modifiability support in regard with questions Q4 to Q5 refers to the previously explained arguments and they are indicated as direct and indirect support for Q4 and Q5, respectively. For scalable transition costs (question Q6), the scenario T4 provides a good example. Redundant infrastructure components are reassigned from the BANK-ACC application to the SALES application. Q6 can be answered in a similar way as Q5. The transition cost of an scalable change can be derived from aggregating costs of individual modifications. In the example, adding the costs of implementing two redundant infrastructure components for SALES and substracting the costs of keeping the same to redundant infrastructure for BANK-ACC.

Functional compliance. Functional compliance, as described in Section B.3 involves the capabilities of a system to adhere to standards, conventions or regulations in laws and similar prescriptions relating to functional suitability. For process and application integration systems, an important aspect of functional compliance is compliance to process regulations [Daniel 2009], [Kharbili 2008] – or compliance process patterns [Ghose 2008]. The identification of non-compliant processes can be a good indicator of possible future failures and increased costs to correct software systems [Lu 2008].

In the second business scenario (B2), it was indicated that process regulations determine that when direct sales agents the return to the bank after visits to their clients, they must report those visits to supervisors and sign a document registering visits. For the *Loan Management* process in Figure 7.1 and its variation in Figure 7.4 the rules indicate that

- The *Visit Client* activity (A) should precedes the *Report to Supervisor* activity (B).
- The *Visit Client* activity (A) should precedes the *Sign Visit Client Doc* activity (C).

If the previous process regulations need to be checked automatically, it would require a mechanism to check if A precedes B and also A precedes C in the initial situation of Figure 7.1 and after changes to the process, which are shown in Figure 7.4. According to the regulations above, it is not relevant if activities B and C are performed in parallel (as shown in Figure 7.7.a) or in sequence (as shown in Figures 7.7.b and 7.7.c) while they always precede activity A. Even if some other tasks are performed between activities (as shown in Figure 7.7.d) the process would be compliant. Exact and inexact pattern matching as described in Chapter 5 can be used to

provide support for this type of process-level pattern compliance checking.

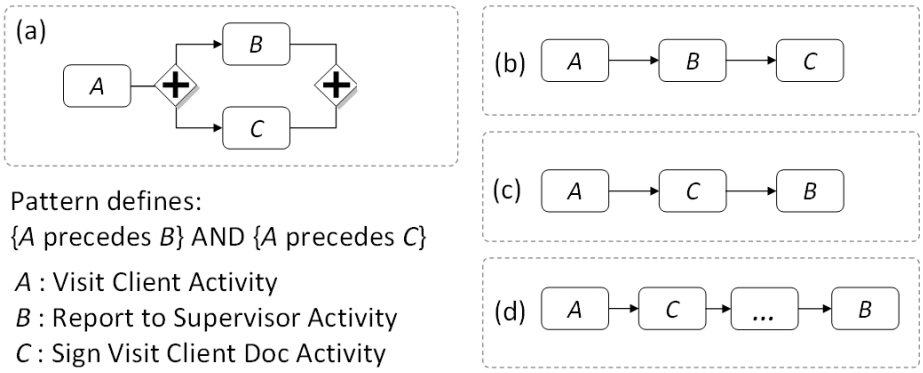


Figure 7.7: Pattern constraining the *Loan Management* process and its variation in Figure 7.4.

Traceability in LABAS can also provide support to functional compliance. In this case, trace links created between pattern roles and their instances (see Section 3.4) are not modified after changes to the *Loan Management* process. Because they are not modified, an evaluation of rules sensing changes in the model does not trigger change actions on trace links, and elimination of modification of pattern roles do no occur, and pattern instances are not eroded. This indicate that the process is still compliant to regulatory patterns. The example illustrated here is simple, but automated support becomes relevant when small and distributed modifications to pattern instances take place on large and interrelated models. The costs of analysing the impact of changes on these pattern instances can be significantly increased with the amount of non-traced elements.

The following questions mention some elemental facilities that tools offering functional compliance support should be able to answer.
Q7: Can process regulations be documented?
Q8: Can compliance pattern instances be identified in actual processes automatically?
Q9: Can be compliance pattern support be quantified?
Answers to these questions using LABAS and a manual-based approach are indicated in Table 7.2.

7.2.3 Electronic Bill Presentment and Payment (EBPP) Case

7.2.3.1 Analysis goals

The previous case study (Section 7.2.2) focused on analysability (traceability and complexity), functional compliance and changeability (modifiability and scalability).

The analysis goal in this case study is to assess the capabilities of the proposed framework to provide improved suitability, robustness-based changeability and reusability. The reason to consider two case studies is twofold, to address a wider business domain perspective (for instance, here it is included an organisational merger) and also to maintain the cases manageable in terms of the architecture and process models illustrated in the figures. Larger and more complex models are less suitable to be presented in a textual form, for these cases the use of appropriate CASE tools is preferred. Similar to the previous case, the LABAS framework is compared to an unsupported (hand-crafted) and more human-centric approach. The differences in effort regarding maintainability (robustness and reusability) are discussed through the different change scenarios. The same scenarios are used to analyse suitability. The change scenarios involve the merger of a number of organisations and a process-wide application architecture migration from a legacy-centric to a service-oriented architecture. The analysis is a post-mortem analysis; therefore, there is no need to normalise the weights of each change scenario [Bengtsson 2004].

7.2.3.2 Processes and software architecture

This case study involves a billing and payment process, common to utility companies. The process is a good example where customers and businesses interact and it represents a typical process in the e-commerce domain [NACHA 2010]. The process describes the presentment of bills from a number of utility companies to their customers. The payment of bills is simplified to a single payment method that consist of a bank transfer. A network of banking institutions is also involved in the payment process. The case illustrates an organisational merger between utility companies. The merger is simplified to focus on one of the critical activities in the process. The activity is implemented differently across organisations. The organisations are identified as a main utility company and two new regional subsidiaries acquired during the merger. The case describes the integration challenges in terms of automating several process steps, integrating isolated applications and merging the processes (and supporting software) from the main utility company and the recently acquired regional subsidiaries.

Figure 7.8 shows a high level view of the bills presentment and payment process. The model in the figure hides the differences across organisations[5]. Three participants (roles) are exhibited at this level: customer, bank network, and utility company. Periodically, a utility company bills customers with an amount of money corresponding to the consumption of the delivered services. Customers receive their bills and decide if they will pay or become indebted. A payment on the due date will

[5]Differences in processes are illustrated later in the section in lower level process models.

eliminate the debt of the customer, otherwise the debt is accumulated. After the payment transaction is completed, the bank network sends the remittance information to the customer and the biller.

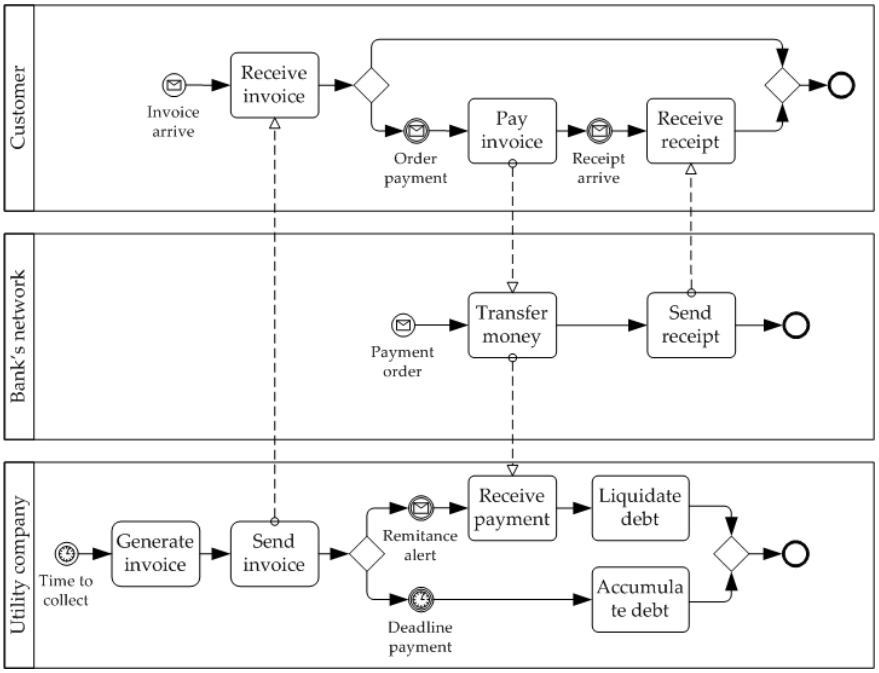


Figure 7.8: Billing and payment process modelled in BPMN [OMG 2008b].

An extract of a domain model with business object connections is shown in Figure 7.9. Figure 7.10 shows the applications supporting the EBPP process in the main utility company. To a great extent, these applications work in an isolated way. The flow of information throughout the process is mostly integrated by means of actions performed by human participants of the process. If there are connections between applications, these are point-to-point connections. The utility company utilises (for the EBPP process) a Customer Relationship Management system (CRM), Enterprise Resource Planning system (ERP) and two custom-built applications for creating invoices (bills) and measuring consumption, the BILLING and METER, respectively. The ERP system manages rules applied to constantly changing tariffs. The METER application manages customer consumption information. The CRM system contains information of customers that is used to maintain the relationship between them and the company. The applications supporting processes in the two recently acquired regional subsidiaries are identified in 7.10 with the participant roles: utility company A and B.

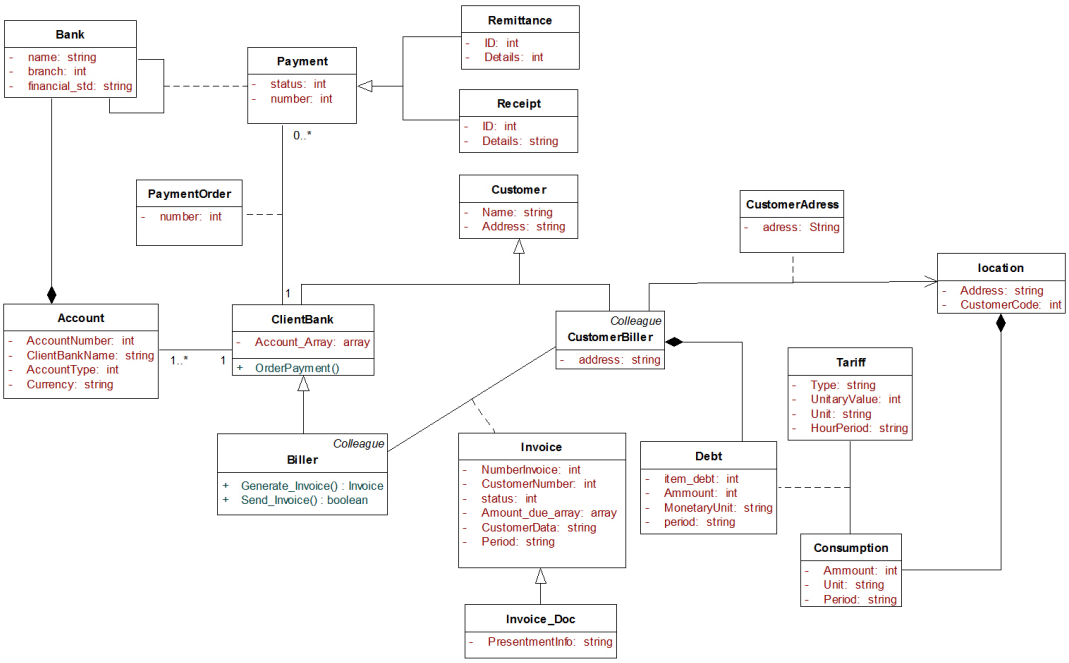


Figure 7.9: Extract of domain model for EBBP process.

7.2.3.3 Change scenarios elicitation

A number of business and technical requirements constitute the basis for change scenario elicitation. The next paragraphs detail these scenarios.

First business scenario (B1). A main utility company acquires two regional utility companies, which become subsidiaries of the main utility company. After the merger, one of the first requirements is to standardise the generation and presentment of bills for customers in all the organisations, while trying to diminish costs and to simplify the set of applications supporting the process. Figure 7.11 illustrates differences for generating bills among the three organisations. The companies' roles are indicated as *utility Company*, *utility Company A* and *utility Company B*, referring to the main utility company and its two new subsidiaries, respectively.

Second business scenario (B2). Central directions from the main utility company have indicated that activities from the subsidiary companies should follow the process structure of the main utility company. Changes to *send bill* and *payment* activities in the main utility company were not desired since corporate contracts with a customer service provider (for invoice presentment) and with the bank network (for bank transfers) should remain unchanged at this stage. However, managers asked to IT architects to analyse the situation with the current process-wide applications

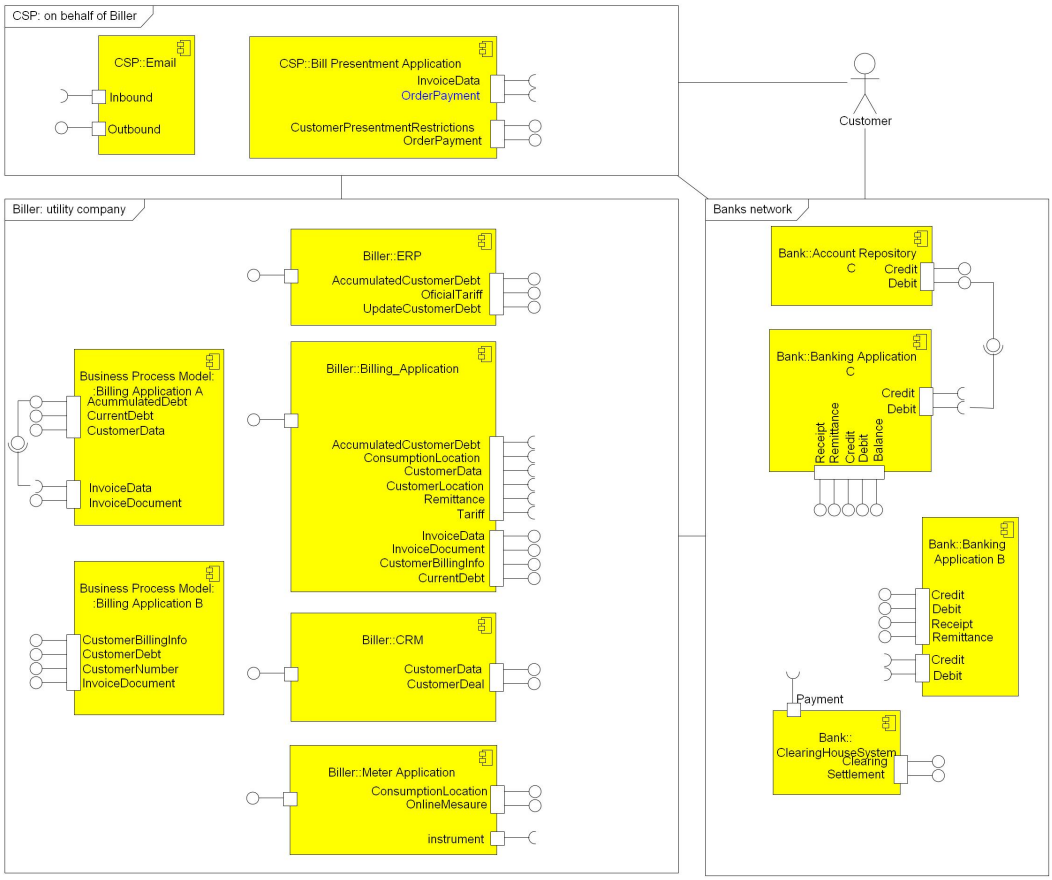


Figure 7.10: Applications associated to different roles participating in the EBBP process.

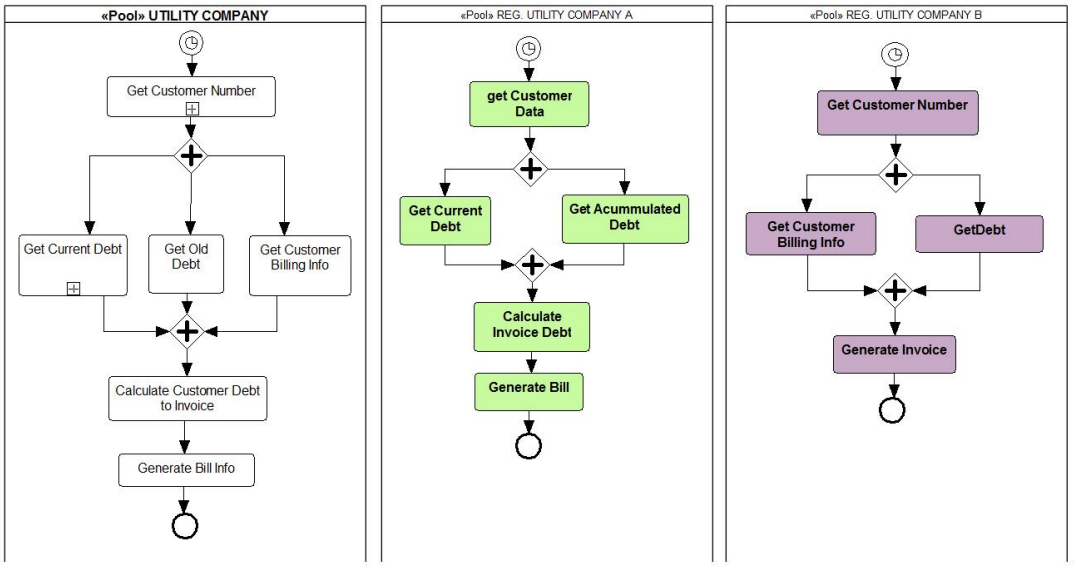


Figure 7.11: Variations between *Generate Bill* activities across utility companies.

architecture. They expected to improve the operation of the entire EBPP process by integrating its supporting applications. Since more organisational mergers were envisioned, it was requested to prepare the process-wide application architecture to future changes at application level, while maintaining the alignment to the main utility company processes.

Technical scenario (T1). In order to respond to the first business scenario, the team of IT architects from the different utility companies have to review the Generate Bill activity and supporting software and to decide how the different applications would be integrated and consolidated. Initially, they decided to move the cost of integration to the *Customer Service Provider* (CSP) presenting final invoices to customers in the main utility company. The company already had the contract with the CSP, and a change would only shift the recipient of invoices sent by the acquired utility companies A and B from final customers to the CSP. The CSP would distribute invoices from all utility companies to all customers.

Technical scenario (T2). In order to respond to the second business scenario, IT architects in the main utility company decided to migrate the current legacy-based application architecture to a service-based architecture. The new architecture would add one level of abstraction between legacy applications and processes. The idea was to abstract processes from changes at the application level, while maintaining alignment to the processes in the main utility company.

7.2.3.4 Scenario Evaluation

After describing the scenarios and associated architectures in the EBPP case, the modification efforts from one scenario to the other are analysed. The analysis focuses on suitability, robustness-based changeability and reusability. Appendix B describes these system sub-characteristics and associated metrics. Table 7.4 summarises the scenario evaluation. The rest of the section describes how these results were obtained and it discusses the advantages of the proposed framework and techniques in comparison to a manual-based approach.

Suitability. Suitability refers to the adequacy of a software system in terms of its coverage of user needs and correctness of implementation. For integration systems, functional suitability focuses on coverage and correctness of integration needs.

An estimation of functional *suitability* can be derived using Equation B.1 (Appendix B). The measure is based on a metric of functional completeness for process-centric integration solutions $fc_{int} = \left(1 - \frac{RIP_i - DIP_i}{RIP_i}\right)$ and the number of applications

Table 7.4: Scenarios evaluation - EBPP case study

	LABAS		DIRECT / MANUAL	
	B1=T1	B2=T2	B1=T1	B2=T2
Suitability	0.2	1	0	0
Reusability	0.3	0.2	0	0
Robustness	+	+	−	−

directly involved with process elements requiring integration $A_{int} = \left(\frac{\sum_j A_j(RIP_i)}{\{A\}}\right)$, with $suitability = fc_{int} \cdot A_{int}$. Process-level integration needs are indicated by data flow elements connecting process elements supported by different application architecture components. DIP captures process elements that require integration but only if they have been explicitly related to their supporting applications. RIP identifies integration needs at process level even if they do not have explicit connections to elements in lower level elements.

In the first and second technical scenarios, a number of integration needs are identified. These involve integration and consolidation of applications supporting the *generate bill* activity, as well as the integration and abstraction from business levels for the rest of applications supporting other activities in the EBPP process.

An unsupported (hand-crafted) and more human-centric approach would quantify the required integration points by manually identifying data flows connecting process steps supported by different applications. Considering the models from Figures 7.8 to 7.11, IT architects would be required to manually identify what data is being processed in each activity, what applications are sources of this data or if they are performing some processing steps on the data. In this manual-based case, DIP is equal to zero, since explicit relations between process models elements and application architecture components do not exist. To obtain a value for the total amount of RIPs, architects would count manually, obtaining $\{RIP\} = 14$ (if false positives and negatives can be avoided). The total amount of RIPs will be clarified later, when the calculation of DPIs is explained. In terms of analysability, the time spent during a manual-based analysis is significantly increased when traceability information is poor or does not exist.

In the LABAS approach, additional information of relations between elements involved in the integration problem is available. Section C.1 in Appendix C provides models in the LABAS's BAIL layer. Using the traceability information from these models, $\{DIP\}$ can be calculated as follows. For every two consecutive process model elements (x, y) connected through a path, it is verified if the number of domain model elements connected to x and y is linked to more than one application.

If the latter is true, then a DIP is quantified for that (x, y) pair. In other words, for every $(x, y) \in E(P)$ and $x, y \in V(P)$, $\{RIP\} = \sum_{i,j}(x, dx_i, A_j) \cup (y, dy_i, A_j)$, with $\{DIP\} > 1$ and $dx, dy \in V(D)$, $A_j \in V(App)$. Note that P, D and App are graphs[6] representing the flattened process model, domain model and application architecture model.

Consider the example of the *generate bill* and *send bill* activities, which are disaggregated into lower level process models in Figures C.3 to C.7 of Appendix C. For these two activities there are *ten* DPIs among flows between activities and applications associated through domain model elements (three DPIs are associated to the *send bill* activity are identified in the first scenario, and the remaining six DPIs in the second scenario). DPIs in these activities are related to application architecture elements by domain model elements as follows.

• Internal flows of *generate bill* activity:

- *get customer number* and *get current debt* → CRM and BILLING through *Customer* and *Debt*.

- *get unitary tariff* and *calculate current debt* → ERP and BILLING through *Tariff* and *Debt*.

- *get customer number* and *get old debt* → CRM and ERP through *Customer* and *Debt*.

- *get customer number* and *get customer billing info* → CRM and BILLING through *Customer*.

- *receive customer address* and *request consumption location* → CRM, BILLING and METER through *Customer* and *Consumption*.

• Flow between *generate bill* and *send bill*) activities:

- *generate bill info* and *get bill info* → BILLING and BILL-PRES through *Invoice*.

• Internal flows of *send bill* activity[7]:

+ *customer service provider* on behalf of *utility Company* (Main):

- *create bill presentation* and *send bill document* → BILL-PRES and EMAIL through *Invoice-Doc*.

+ *Utility Company A*:

- *create bill presentation* and *send bill document* → BILLING-A and EMAIL through *Invoice-Doc*.

+ *Utility Company B*:

- *create bill presentation* and *send bill document* → BILLING-B and EMAIL through *Invoice-Doc*.

For the entire EBPP process, there are *four* additional DPIs related to *payment* and *debt liquidation/accumulation* activities. They involve the BANKING APPLICATION B, C and CLEARING HOUSE system in the bank's network role, the BILL-PRES (bill presentment application) in the customer service provider role (on behalf of the *utility company*), and the ERP system in the *utility company* role. Figure 7.12 illustrates graphically the *fourteen* identified DIPs, three for the first scenario and

[6] For details on the graph-based notation see Chapter 4.

[7] Note that utility companies A and B have been sending invoices through conventional mail services. After the merger, this service is migrated to email notifications. DPIs between billing applications in subsidiary companies (BILLING-A,B) and the EMAIL in CSP were also considered.

the remaining ones in the second scenario. Using these figures and Equation B.1, an estimated value of *suitability* in $B1 = T1$ - in relation to the final RPIs in $T2$ - can be calculated as $[1 - (3 - 14)/14] \cdot [14/14] \approx 0.2$, while for $T2 = B2$ suitability would be equal to one. For a manual-based approach and no links relating process elements to their supporting applications, suitability would be considered equal to zero.

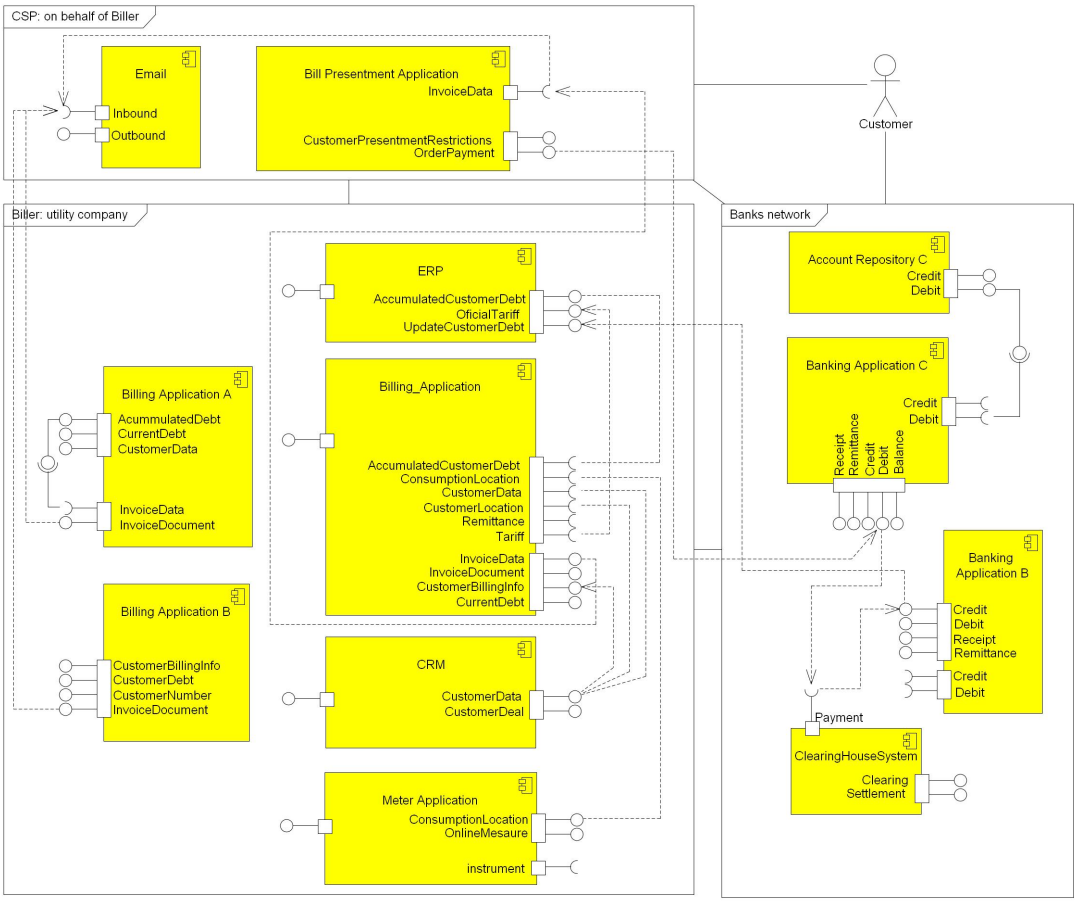


Figure 7.12: DIPs at application architecture level.

Reusability. According to the standard ISO/IEC 25010, reusability refers to the degree to which an asset can be used in more than one software system, or in building other assets. As described in Appendix B, for large and distributed organisations, a common situation in practice is that software systems created to solve a similar problem, but in different times and locations, often do not share their design solutions. The latter can be observed in a simplified way in scenarios B1,T1 where three different approaches to generate and present bills were created in the main, A and B utility companies.

As described in scenario T1, IT architects from the different utility companies de-

cided to tackle the integration problem from the customer (when bills are presented) to then go backwards until bills in the three different organisations are generated. A first solution considered the utility companies A and B sending their bill documents to the Customer Service Provider (CSP) role (Figure C.7).

After requirements in the second scenario (B2,T2) a migration from the previous legacy-based application architecture to a service-based architecture was adopted. Two approaches were considered to design new services. One takes each activity in the EBPP process and consider them as service candidates. The other encouraged the reuse of documented process-level design knowledge to design services, in this case according to standard bill presentment and payment activities.

The first approach tries to illustrate service design strategies that directly translate business process level descriptions to process-centric service descriptions, which are later refined to meet technical requirements, e.g. [Gardner 2003], [Ouyang 2009]. A shortcoming of these approaches is the assumption that software services have a one-to-one relationship with activities in business process models; however, software services can involve more than one business activity or can even be more granular than a business activity [Koehler 2006].

Also, possible redundancies at process model level should be reviewed before considering any refinements to service levels. As several authors have indicated, business level integration should precede technical level integration [Puschmann 2004], [Janssen 2005] and [Lam 2005]. Reuse at business level can be illustrated with the organisational merger in the first scenarios. In this scenario the same business operation (generation of bills) is implemented in three different process variations (Figure 7.11). If three services for generating bills would be directly derived from models in Figure 7.11, they would likely become redundant. In this example, the pattern discovery technique proposed in this work (Chapter 6) can be used to discover frequent process sections in large and complex models. In this case, there are several models in different levels of abstraction involved. After flattening the models into the lowest level, the pattern discovery technique can be applied to find inexact and partial structures that frequently occurs in models. For the three different versions of the *Generate Bill* activity, this could be done adjusting the size of expansion steps of the pattern discovery algorithm (see Section 8.6.1 for an example of how the algorithm works) and utilising an approach of semantically enhanced vertex matching (see Section 8.5 for an example).

The second approach to design services is based on a standard reference model and associated process-level patterns for Electronic Bill Presentment and Payment (EBPP) in [NACHA 2010]. The reference model included composite activities for *bill creation*, *bill presentment* and *payment*. A biller role, a customer role and intermediaries between customers and billers are the relevant roles in the reference model. Due

to the central role of *customer* and activities *bill creation*, *bill presentment* and *payment* in the case study, the IT architects considered them as candidates to be implemented as services.

The three different implementations of the *bill creation* activity in the three utility companies are analysed before designing a service that offers the results of their operation. In this case, instead of using the pattern discovery technique, as described above, the pattern matching technique can be used to compare the standard description of the *bill creation* activity in the EBPP reference model with the three implementations shown in Figure 7.11. Also, the standard definitions of processes (process patterns) in the EBPP reference model could be compared to processes in the main utility company. The *transfer money* activity (Figure 7.8) can be related to the bank transfer option among the *payment* activities in the EBPP reference model. In this case, inexact pattern matching could assist the analyst or architect by relating the *payment* in the EBPP reference model, that contains more abstract concepts, to the specific payment process made through a bank transfer in the case study.

Note that in large models, analysts and architects may not know that there is more than one implementation of an activity. Using the patterns in a reference model to look for possible instances can also help identify possible redundancies in process models.

The payment activity in EBPP is decomposed into three lower level activities that describe the *payment* initiated by the customer, a *clearing* activity (performed to manage taxes and other charges related to transactions between financial institutions) and the *settlement* of bank accounts for each participant in bank transfer transaction. Based on these activities, three services were defined to create a composed *payment* service: a *transfer* service, a *clearing* service and a *settlement* service. The particular implementations of these services were related to the actual process steps and software support described for the case.

The *customer* service in the final solution is a data-centric service that abstracts customer information from different data sources in the biller side. In order to know where this information is located and processed, explicit trace links relating applications to the *customer* concept are utilised. In this case, the applications are the CRM, ERP, BILLING and METER applications.

Two technical services, *tariff* and *meter* services were designed to encapsulate the rules for tariffs embedded into the ERP and the information regarding customer consumption managed in the METER, BILLING-A and BILLING-B applications. Even though the final invoice documents are generated with the two last applications, they may be replaced in the future with applications used by the main utility company. The replacement would avoid functional redundancy at application level.

Figures C.13 to C.17 in Appendix C illustrate the models in the LABAS BAIL

layer (process model, domain model and application architecture elements related) with their associated new services. The resultant architecture composed of application and service architecture elements is illustrated in Figure 7.13. So far, the *presentment* and *meter* services were reused in the *generate bill* and *send bill* activities for each utility company, resulting in 11 services composing the final service architecture for the EBPP process (Figure 7.13). If separate services for bill generation and presentment in the three utility companies were considered, this number would rise to 15, which is the total service utilisation in the EBPP process. Using Equation B.5, an estimated value of reusability for each of the two services is as follows, $reusability_i^j = (\sum_k [s_k | s_k = s_i, s_k \subset P_j]) / (\sum_k [s_k | s_k \subset P_j]) = 3/15 = 0.2$, with $i = presentment, meter$ services and j the EBPP process. For scenario T1, a single *meter* service was used instead of three services, hence reusability in T1 using the expression above would be $1/3 \approx 0.3$.

In this case study, new services were created based on the reuse of process-level design knowledge from the EBPP reference model and associated patterns. Patterns at lower level can also benefit the architecture solution depicted in Figure 7.13. For instance, instead of maintaining separate messaging communication between services from the utility company and other participant roles, a central element managing communication and messages routing could be incorporated. This is addressed by the well-known Enterprise Service Bus (ESB) pattern [Zdun 2006], [Erl 2008]. In [Gacitua-Decar 2008a], the application of the ESB pattern to the architecture in the EBPP case study is described. The result is the architecture illustrated in Figure 7.14. Note that to simplify the figure, applications that do not interact with customers are not shown. This example attempts to illustrate that reusing design knowledge in the form of patterns can improve architecture design quality, in this last case, improved maintainability. Chapters 5 explains how a technique for pattern matching can be used to identify partial instances of patterns in a target (architecture) model. The model can be modified to complete partial instances with the entire pattern configuration. Also, the description of a frequent problem as a pattern could be used to identify whether the problem is occurring in the target model, and later, to study solutions that can already be available (e.g., the solutions to anti-patterns in [Koehler 2007]).

Robustness (sub-characteristic of changeability). Robustness refers to the capability of a system to be insensitive under changing conditions regarding its perceived value. Appendix B explains that passive value robustness can be achieved by systems that may have excess capability or a large set of latent value, thus increasing the likelihood of satisfying new requirements without changing the system [Ross 2009]. In the EBPP case study, the rationale behind the design of services based on previously

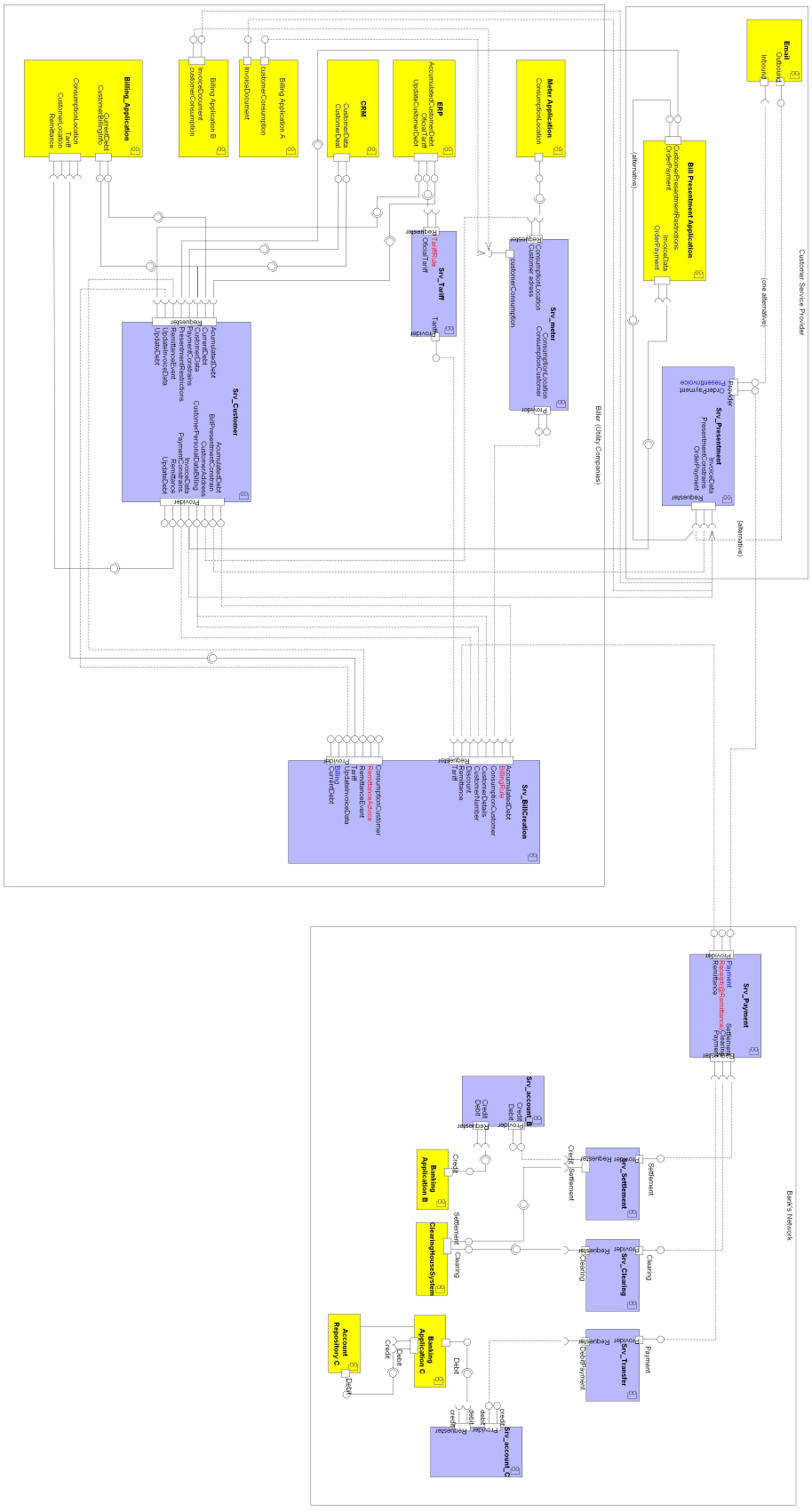


Figure 7.13: Applications and services architecture.

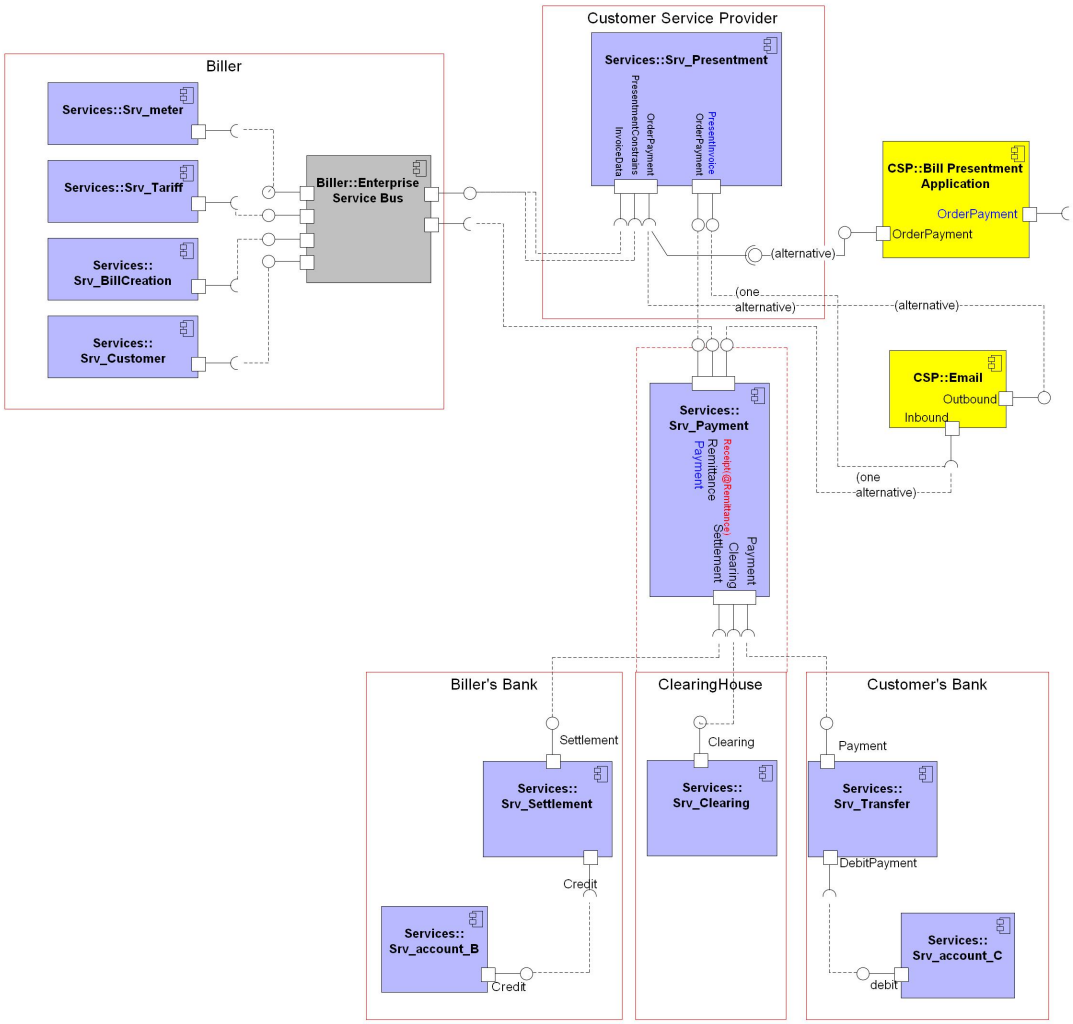


Figure 7.14: Service architecture with instantiated ESB pattern.

documented process-level pattern aims to provide a more robust service design (i.e., defining more stable business-level services that can rely in other lower level, more technical, services). These lower level services can be modified more often and have a greater active value robustness [Ross 2009]. The separation of business level and more technical level services follows the principles pursued by service-based architecture patterns such as the service *virtualization* pattern in [Erl 2008] and service *layers* pattern in [Zdun 2006]. These patterns implement separation of concerns and benefit reuse of services in upper layers of abstraction. The objective is to reduce change associated problems of conventional service design. Services that follow a bottom-up design approach often fulfill particular project requirements within a domain. When any of these services becomes a candidate for reuse in a different context, it usually requires modifications or extensions. Also, services that follow a top-down design approach often must be changed (specialised) to fit in particular contexts. Pattern-based service designs for abstract level services would have increased robustness.

In Table 7.4, robustness is qualitatively assessed and compared to an approach were services are directly[8] defined from activities in process models (e.g., in [Gardner 2003], [Ouyang 2009]). An explicit contribution to robustness is indicated with a $+$ symbol, suggesting that pattern-based design is reinforced. A neutral contribution is indicated with a $-$ symbol, suggesting that designs with robustness in mind would rely on the IT architect's criteria.

7.3 Tool Support

7.3.1 LABAS Profile

A UML profile was developed to create concrete models using the abstract syntax specified for models in the different layers of the LABAS framework. Models can be created in a standard UML tool. Modelling elements of each layer are organised in different toolboxes. The BML toolbox allows the creation of business processes using the BPMN notation [OMG 2008b] and domain models using the standard UML notation [OMG 2007]. Application architectures can be created using the AAL toolbox and service architecture models can use the SAL toolbox or alternatively the profile for Service oriented architecture Modeling Language (SoaML) [OMG 2009b] for the UML modelling tool - Enterprise Architect [SparxSystems 2010]. Trace links, which follow the traceability metamodel specified in chapter 3, use the TRACES toolbox. Figure 7.15 shows a snapshot of the LABAS toolboxes implemented in a

[8] Indicated in Table 7.4 as DIRECT.

standard UML modelling tool - Enterprise Architect [SparxSystems 2010] and Figure 7.16 shows the graphical specification for toolboxes in the framework.

Pattern configurations associated to a specific layer and model perspective are modelled with the same notation adopted to models in that layer and perspective. Additional information is captured in the pattern template as described in Section 3.2.3. Pattern templates can be implemented in the same tool used to implement the LABAS profile [SparxSystems 2010]. Several pattern templates can be organised in a pattern repository and instantiated on models. Figure 7.17 shows a snapshot of a pattern template implemented in Enterprise Architect [SparxSystems 2010].

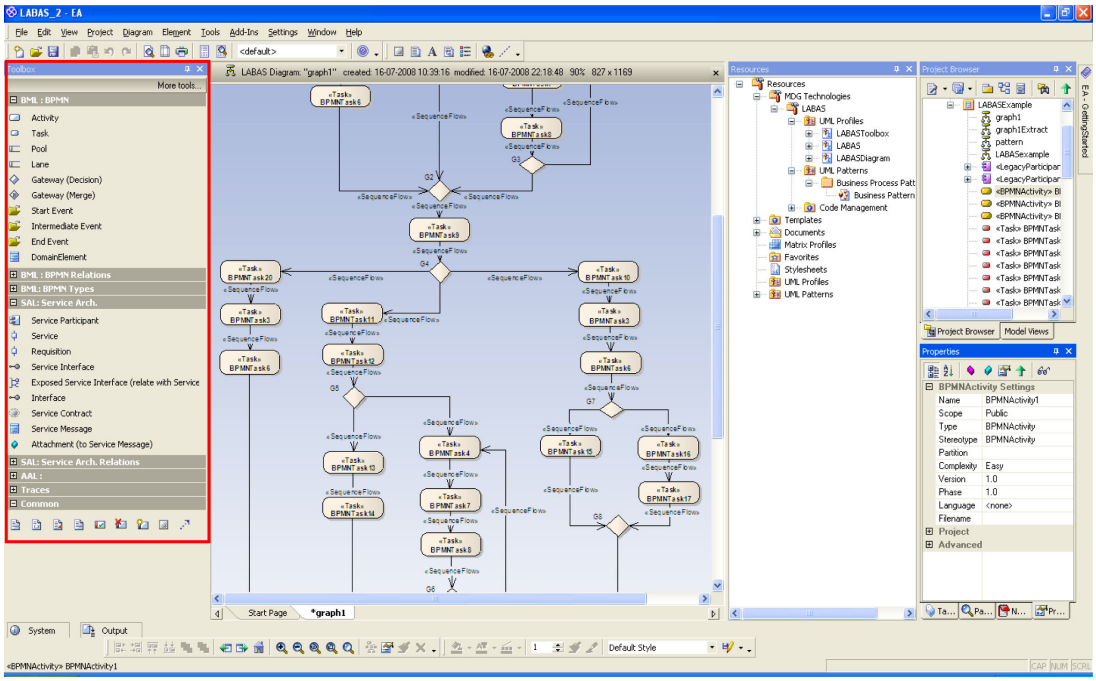


Figure 7.15: Snapshot of LABAS toolboxes in standard UML tool

7.3.2 Model to Graph Transformation

Process models and patterns created with the LABAS profile and pattern documentation support in a standard CASE tool [SparxSystems 2010] can be exported as XML (XMI 2.1) documents, which can afterwards be transformed to files describing graphs used as input for the pattern matching and discovery techniques in LABAS. Figure 7.18 shows a simple transformation from the schema used to store process models with the LABAS profile to two data structures storing the adjacency matrix and label vector providing the information to describe process graphs representing process models. This particular transformation was created using Altova Mapforce 2008 r.2 [Altova 2008]. For composite process models a previous flattening step is required.

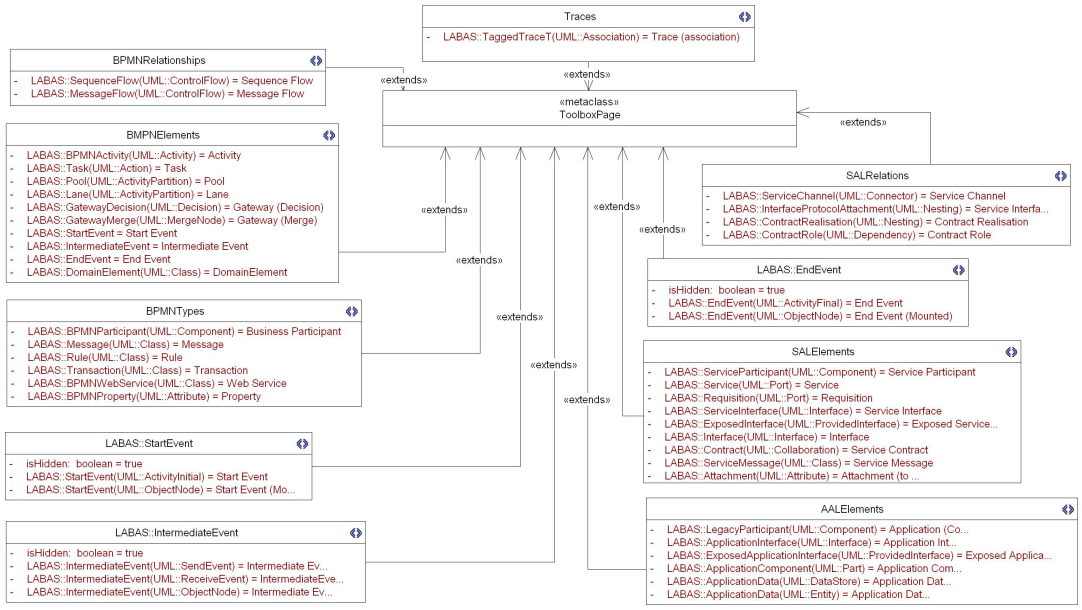


Figure 7.16: LABAS toolboxes diagram

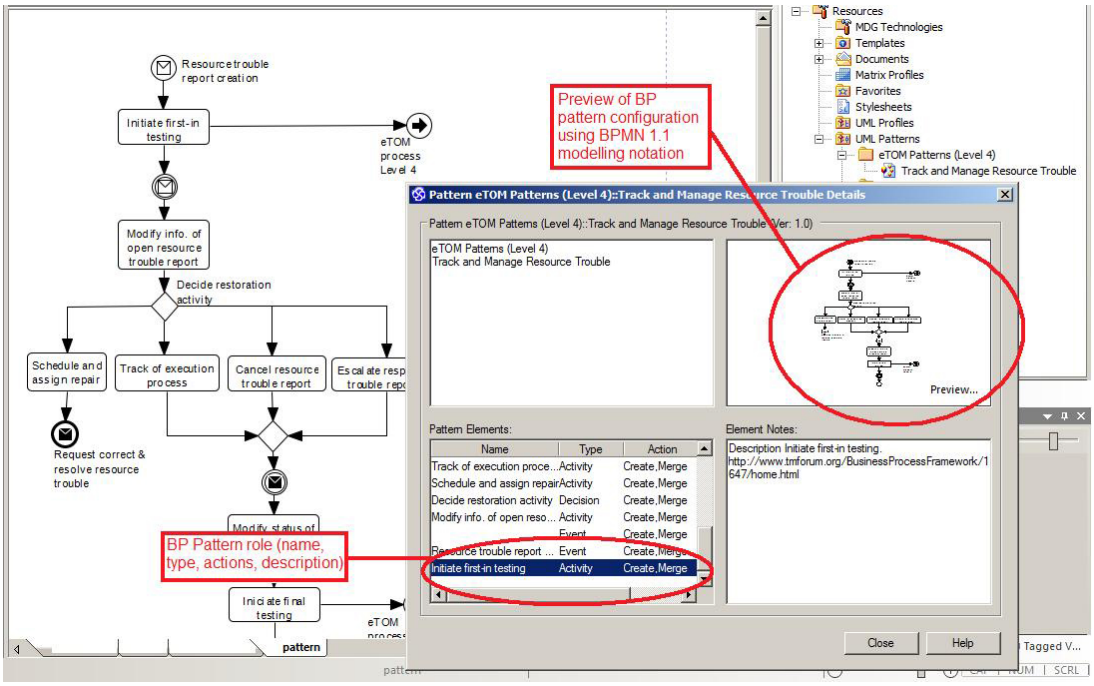


Figure 7.17: Snapshot of pattern template in standard UML tool

An automated flattening facility has not yet been fully implemented and is planned to focus on the next version of the business process modelling notation (BPMN v2.0) (which is compliant to the future version of the Business Process Definition Metamodel – BPDM 1.x, specified by OMG). More details on the current LABAS profile in Appendix section 7.3.1.

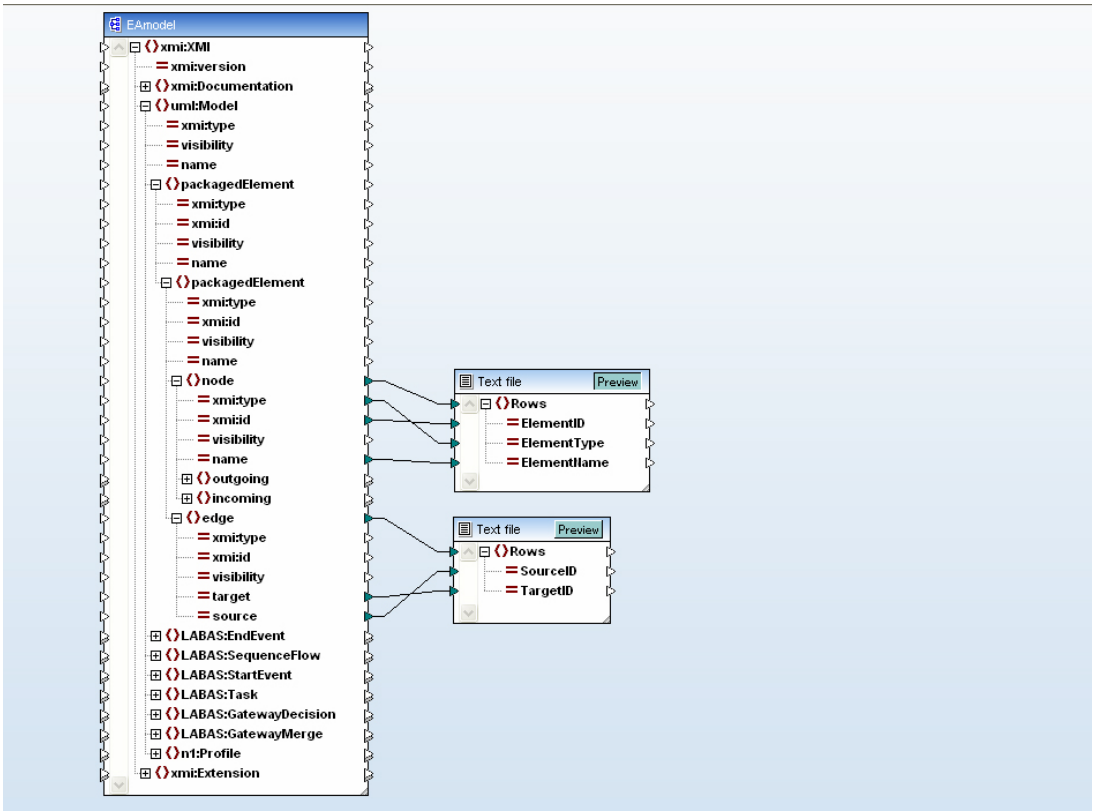


Figure 7.18: LABAS process model to graph adjacency matrix and label vector transformation.

7.4 Summary

The case studies presented in this section were used to illustrate the benefits of the proposed framework and techniques in relation to maintainability and functional suitability and compliance. A the beginning of the chapter (Section 7.1.3), a number of challenges (problems) P1 to P5 affecting maintainability and functional suitability and compliance were explained. The scenarios in this chapter illustrate how IT architects could use the modelling and pattern-based support provided in the LABAS framework to tackle challenges P1 to P4. P5 (which is related to challenges regarding processing time and errors) is addressed in the next chapter, where the effectiveness

and efficiency of techniques to automate pattern matching and discovery are evaluated.

The problems associated to maintaining separate models for process, domain and architecture descriptions (P1) is addressed with modelling support for interconnected layers of the LABAS framework (Chapter 3). Corrections in models due to misalignment between process and architecture levels after changes (P2) is assisted through explicit trace links of the traceability model (Section 3.4). Ineffective or inefficient use of design knowledge (P3) is addressed by encouraging pattern identification and application. Automated pattern identification is supported with pattern matching and discovery techniques (Chapters 5 and 6). Lack of automated support for checking regulatory process compliance (P4) can be addressed with the help of the pattern matching techniques by comparing actual models against regulatory patterns.

As any effort emphasising architecture analysis, costs associated to models and architectural abstractions documentation and maintenance are increased. These costs should be balanced in the context of the integration problems being addressed. Dealing with simple and small models may only require a hand-crafted and human-centred approach. However, integration of processes and applications in large and distributed organisations can be benefited by an explicit and automated support to analyse architectures, models and associated abstractions. The simplified examples in the case studies presented in this chapter attempted to illustrate that even simple models can be complex to analyse. In real scenarios, automated support and an organised framework can be critical for a more suitable, compliant and maintainable integration solution.

Chapter 8

Evaluation of Matching and Discovery Techniques

Contents

8.1 Overview

The previous chapter focused on the overall framework and its architectural and modelling support during the development of process-centric and service-based integration systems. This chapter describes an experimental evaluation that attempt to measure the effectiveness and efficiency of the proposed pattern matching and discovery techniques. The chapter also contains substantial illustrations (including case studies) regarding the feasibility, applicability and practical relevance of the techniques. Tool support associated to the implementation of the techniques and the environment for the experimental evaluation are described towards the end of the chapter.

8.2 Definition and Planning

The experimental evaluation described in this chapter considers a strategy for a single product and team [Basili 1996], where a researcher is interested in better understanding the quality of a product (software/technique) using an *a priori* set of variables for observation. The evaluation considers a number of experiments and cases to investigate the effectiveness and efficiency of the proposed pattern matching and discovery techniques in different conditions of model and pattern graphs. The aspects of the evaluation include

- *Effectiveness.* An evaluation of the techniques' effectiveness in terms of accuracy in the obtained results when using the algorithms described in Chapters 5 and 6. Trends in accuracy measures during the matching and discovery process are studied while increasing the size and other parameters of model and pattern graphs.
- *Efficiency.* An evaluation of the pattern matching algorithm's processing time. Trends in the processing time obtained from experimental results using randomised model and pattern graphs of different sizes are used to experimentally estimate the time complexity of the algorithm.
- *End user intervention.* Tool support for the involvement of end users in the process to match and discover patterns in model graphs is described. This includes the implementation of a UML profile and functionality to transform a process model description (using the profile in a standard CASE tool) to the generic graph representation used in the implemented prototype tool. Also, pattern documentation support is illustrated.

8.2.1 Type of Experimental Evaluation

A comparative experimental evaluation assessing the strengths and weaknesses of different techniques requires a common corpus of freely available models [Sartipi 2001], [Koschke 2000]. Unfortunately, to the best of the author's knowledge, that is not the case for matching and discovery techniques for process-centric descriptions (e.g., process-centric service architectures and business process descriptions). Private corpuses such as the SAP reference model were excluded from this work due to difficulties in accessing the data. Instead, an alternative experimental evaluation involving the techniques and a pre-validated benchmark was used. The benchmark consists of a set of pre-defining models with known instantiated patterns. The experimental evaluation aims to explore the effectiveness and efficiency of the matching and discovery algorithms used in the proposed techniques.

- Effectiveness of the matching and discovery algorithms is studied by measur-

ing accuracy of their results while varying specific parameters (e.g., the number, size and structure of instantiated patterns). Accuracy of the results is expressed in terms of recall- and precision measures.

- Efficiency is studied measuring the algorithms' processing time throughout a set of tests using randomly generated process graphs.

Process graphs used to study the efficiency of algorithms are more structured than purely random graphs. They have source and sink vertices and vertices with a bounded in/out-degree, common to process models [Golani 2003]. Graphs with cycles and non-free of deadlocks are allowed. The latter attempts to represent process-centric models that can contain errors if they are executed. This can occur in models available in early stages of systems development, which are the stages of interest in this work.

A correlational study varying relevant parameters associated to model and pattern graphs is used to explore trends in the quality of results. Quality of results refers to accuracy and time complexity of the graph matching and discovery algorithms. Among the manipulated parameters are the number of pattern instances, pattern overlaps, size and in/out-degree of vertices in model and pattern graphs, the structure of patterns and semantic characteristics of graph vertices.

When human intervention is involved in the study of a (semi-automatic) technique, costs and organisational constraints can be the causes of a poor experimental setting. Experimental studies in software engineering often involve a very small number of subjects, resulting in experiments with insufficient statistical power to investigate variations and obtain accurate measures. A limited but manageable experimental environment is used here to emulate human intervention during the utilisation of the proposed techniques. This intervention is modelled through defining variations in parameters that are available to end users in the implemented prototype tool. In particular, these parameter variations capture changes in threshold levels for semantic matching and the number of elements for pattern instances to be discovered. When semantic matching includes natural language processing techniques, this thesis refers to a demonstrator case described in [Gacitua-Decar 2009a] and revisited in Section 8.5. A comprehensive evaluation of this (semantic) aspect, which can involve natural language processing techniques, is out of the scope of this work. For similarity between vertex labels, specific label (sentence) similarity techniques investigated in [Li 2003], [Li 2006] were adopted in this work. Here, their results are used as a reference to the quality of the sentence similarity techniques that can be used in the proposed algorithms.

8.3 Experiments - Matching Graph Structure

Positive and negative sample subgraphs were used to test the matching technique on model graphs with different sizes. A positive sample is a subgraph that always occurs in the model graph and is referred to as a positive pattern sample. Similarly, a negative sample is a subgraph that never occurs in the model graph and is referred to as a negative pattern sample. The matching technique was tested using positive and negative sample patterns of small size (between three and nine vertices) and model graphs of different sizes (i.e., 10, 50, 100 and 1000 vertices). An extract of the samples and model graphs is shown in Appendix D.

In order to generate positive and negative sample patterns, the function `genPosSample` and `genNegSample` were used (see Appendix D). The former creates a subgraph of a model graph centered on a specified vertex and expanded with its neighbours a determined number of times to create the positive sample. The latter derives all connections that do no exist between vertices in a model graph and extracts a subset of a specified size. This subset and a number of non-existent connections define a negative pattern sample. An example of positive and negative samples for a model graph with 10 vertices is shown is Figure 8.1. In this example, vertex matching is simplified to label matching, where labels are numbers. The three vertex labels of the positive pattern sample and their structure are matched in the model. In the graph model, the instance of the positive sample is highlighted in green. The number of vertices and edges included in the instance is shown in brackets (six). A number one between brackets indicates that only the label of a vertex was matched. This is the situation of the negative sample, where only the vertices are matched but not the entire structure of the negative sample.

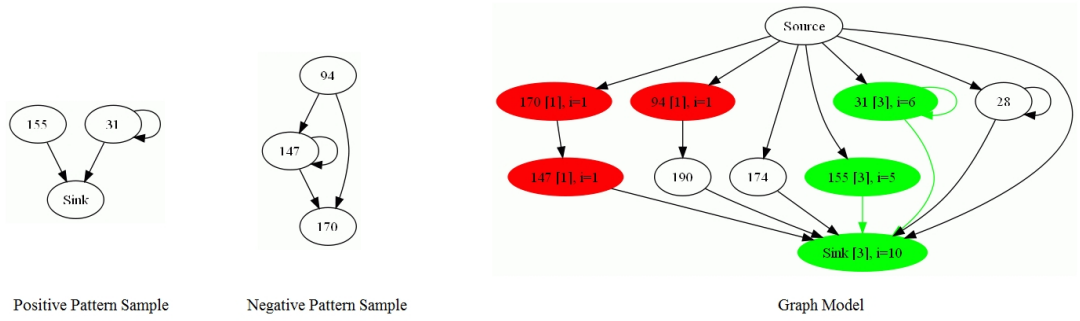


Figure 8.1: Graph Model with instances of positive and negative pattern samples.

Directed and undirected versions of positive and negative sample patterns were matched with directed and undirected versions of model graphs, respectively. The aim of the test is to estimate how effective the matching algorithm is by taking ran-

dom sample patterns and verify if the results are relevant or not. Effectiveness of the automated matching technique in comparison to a manually performed matching can be associated to the effort that an end user has to do to modify a match result into an intended result, i.e., a correct result. In [Melnik 2002], the authors introduced a *match accuracy* metric to test a proposed schema matching algorithm. The metric can be adopted for pattern matching in process graphs. The metric estimates the effort of an end user to modify a pattern match result into an actual pattern instance in the model graph. The *match accuracy* metric (mA) adopted from [Melnik 2002] and the typical accuracy (A) metric are described, as follows.

$$A = \frac{t_p + t_n}{t_p + f_p + f_n + t_n} \tag{8.1}$$

$$mA = 1 - \frac{(n-c) + (m-c)}{m} = \frac{c}{m}\left(2 - \frac{n}{c}\right) \tag{8.2}$$

$$mA = Recall\left(2 - \frac{1}{Precision}\right) \tag{8.3}$$

Where,

- m indicates the number of operations that an end user has to perform to manually identify a pattern instance and no mistakes are done. The result of these operations define the intended result of a match, i.e. $(t_p + f_n)$. Each operation would check if a vertex in the model graph has to be considered as part of a correct matched pair. This pair contains a vertex in the pattern sample (pattern role) and a checked vertex in the model graph that correspond to an exact pattern role instance from a complete pattern instance.
  m can be seen as a function of the model graph size, pattern graph size and maximum out-degree ($maxOutDeg$) of vertices in the model and pattern graph. In the worst case, an end user would check all neighbours of vertices in pattern and model graphs for each vertex in the model graph matched to a vertex in the pattern graph, i.e., $([maxOutDeg(v) \cdot maxOutDeg(u)] \cdot [V(P) \cdot V(M)])$, with $u \in V(M)$, $v \in V(P)$ and $V(M), V(P)$ the sets of model and pattern graph vertices. It was considered that the effort to match all pairs of pattern role and pattern role instances from pattern and model graph vertices is $V(P) \cdot V(M)$ vertices – the worst case scenario.
- c refers to the number correctly suggested vertices in the result of a structural match, i.e., correctly identified pattern role instances (true positives t_p).
- n refers to the number of all vertices suggested in a structural match result, either if the are correct or incorrect suggestions (true positives plus false positives $[t_p + f_p]$).

- $(m - c)$ indicates the number operations that an end user has to perform to add the missing vertex matches to the structural match result (false negatives f_n).
- $(n - c)$ indicates the number of operations required to remove erroneous matches from the structural match result (false positives f_p).
- P (*Precision*) refers to the fraction of correct matched pairs (pattern role, pattern role instance) among pairs from match results (i.e., c/n that relates to $\frac{t_p}{t_p+f_p}$).
- R (*Recall*) refers to the fraction of correct matched pairs (pattern role, pattern role instance) among pairs of intended match results (i.e., c/m that relates to $\frac{t_p}{t_p+f_n}$).

Table 8.1 presents the results of matching different pattern samples in a set of model graphs. The results are calculated considering ***exact and complete*** *matches* (i.e., a true positive is counted only if the matched vertex belongs to an exact and complete pattern instance). Exact and partial matches may be detected, but in this test they are counted as false positives. Note that partial matches in other tests can be considered true positives if the intention is, for instance, measuring accuracy for exact complete and exact partial matches. If matched vertices from partial instances are considered true positives and surjection is allowed, then the accuracy for all samples in this test would be equal to one. The latter is represented in column $mA(c/p - E/S)$ of Table 8.1, which shows the values for the match accuracy metric considering as true positives not only complete instances but partial instances and instances whose elements can hold a surjective relation with elements in the pattern sample.

As described previously, models are directed versions of random process graphs with sizes varying from 10 – 1000 vertices. Considering observations from real process descriptions and characteristics of process models synthetically generated in studies such as [Golani 2003], reasonable values for the maximum and minimum in/out-degree of vertices are between four and one. This values were considered when creating random graphs in this and other tests described in the chapter. Vertices have only a label associated since the test is intended to reflect the estimated accuracy of the structural matching and not the vertex matching. Vertex matching, including semantic label matching is demonstrated in a subsequent section. Labels in graph vertices were assigned randomly (uniformly distributed) from a set of 200 numeric labels (1,..., 200).

Results of the matching are obtained using the `testSample` function, which provides results for the directed and undirected versions of sample and model graphs. Undirected versions consider edges of the directed graphs as vertices of the undirected graph versions. The `testSample` function uses the pattern matching algorithm

Table 8.1: Experimental evaluation of structural matching – directed and undirected graphs

DIRECTED MATCH								m	(n-c)	(m-c)	mA	mA	tp+fn	tp+fp	tp	fp	fn	R	P	A	
	Type	sample	ID test	V(M)	V(P)	mo(M)	mo(P)	effort(tp+fn)	effort(fp)	effort(fn)	c-E / noS	c/p-E / S						c/m	c/n		Observations
	POS	1	1	10	3	4	2	240	0	0	1	1	3	3	3	0	0			1.000	
	POS	2	2	10	3	4	1	120	0	0	1	1	3	3	3	0	0			1.000	
	POS	3	3	10	3	4	1	120	0	0	1	1	3	3	3	0	0			1.000	
	NEG	4	1	10	4	4	3	480	192	0	0.6	1	0	4	0	4	0			0.800	
	NEG	5	2	10	6	4	3	720	432	0	0.4	1	0	6	0	6	0			0.727	
Accumulated M 10				10				1680	624	0			9	19	9			1	0.474		
	POS	6	1	50	3	4	2	1200	48	0	0.96	1	3	5	3	2	0			0.706	
	POS	7	2	50	4	4	2	1600	32	0	0.98	1	5	6	5	1	0			0.923	1 surjection - no error
	POS	8	3	50	3	4	1	600	0	0	1	1	3	3	3	0	0			1.000	
	NEG	9	1	50	5	4	3	3000	300	0	0.9	1	0	5	0	5	0			0.706	
Accumulated M 50				50				6400	380	0			11	19	11			1	0.579		
	POS	10	1	100	6	4	1	2400	120	0	0.95	1	6	11	6	5	0			0.792	
	POS	11	2	100	4	4	2	3200	32	0	0.99	1	4	5	4	1	0			0.929	
	POS	12	3	100	4	4	1	1600	16	0	0.99	1	4	5	4	1	0			0.929	1 structural error
	NEG	13	1	100	5	4	3	6000	360	0	0.94	1	0	6	0	6	0			0.684	
Accumulated M 100				100				13200	528	0			14	27	14			1	0.519		
	POS	14	1	1000	3	4	1	12000	120	0	0.99	1	3	13	3	10	0			0.714	
	POS	15	2	1000	9	4	1	36000	792	0	0.978	1	9	31	9	22	0			0.532	1 structural error
	POS	16	3	1000	6	4	2	48000	1632	0	0.966	1	6	40	6	34	0			0.424	
	POS	17	4	1000	5	4	1	20000	480	0	0.976	1	5	29	5	24	0			0.510	
	NEG	18	1	1000	5	4	3	60000	2160	0	0.964	1	0	36	0	36	0			0.410	1 partial match (2 v)
Accumulated M 1000				1000				176000	5184	0			23	149	23			1	0.154		
UNDIRECTED MATCH								m	(n-c)	(m-c)	mA	mA	tp+fn	tp+fp	tp	fp	fn	R	P	A	
	Type	sample	ID test	V(M)	V(P)	mo(M)	mo(P)	effort(tp+fn)	effort(fp)	effort(fn)	c-E / noS	c/p-E / S						c/m	c/n		Observations
	POS	1	1	27	6	8	2	2592	0	0	1.000	1	6	6	6	0	0			1.000	
	POS	2	2	27	5	8	2	2160	0	0	1.000	1	5	5	5	0	0			1.000	
	POS	3	3	27	5	8	2	2160	0	0	1.000	1	5	5	5	0	0			1.000	
	NEG	4	1	27	13	8	5	14040	2080	0	0.852	1	0	4	0	4	0			0.800	
	NEG	5	2	27	15	8	3	9720	2160	0	0.778	1	0	6	0	6	0			0.727	
Accumulated M 10				27				30672	4240				16	26	16			1	0.615		
	POS	6	1	133	6	8	2	12768	192	0	0.985	1	6	8	6	2	0			0.375	
	POS	7	2	133	8	8	3	25536	576	0	0.977	1	8	11	8	3	0			0.800	1 surjection - no error
	POS	8	3	133	5	8	2	10640	0	0	1.000	1	5	5	5	0	0			1.000	
	NEG	9	1	133	11	8	3	35112	1584	0	0.955	1	0	6	0	6	0			0.667	
Accumulated M 50				133				84056	2352				19	30	19			1	0.633		
	POS	10	1	272	11	8	5	119680	2200	0	0.982	1	11	16	11	5	0			0.881	
	POS	11	2	272	8	8	3	52224	192	0	0.996	1	8	9	8	1	0			0.929	
	POS	12	3	272	7	8	3	45696	168	0	0.996	1	7	8	7	1	0			0.929	1 partial match
	NEG	13	1	272	11	8	3	71808	1584	0	0.978	1	0	6	0	6	0			0.684	
Accumulated M 100				272				289408	4144				26	39	26			1	0.667		
	POS	14	1	2723	5	8	2	217840	800	0	0.996	1	5	15	5	10	0			0.714	
	POS	15	2	2723	17	8	4	1481312	12512	0	0.992	1	17	40	17	23	0			0.521	21 par.m. (1-3v, 20-1v)
	POS	16	3	2723	12	8	6	1568448	19584	0	0.988	1	12	46	12	34	0			0.424	34 par.m. (34-1v)
	POS	17	4	2723	9	8	2	392112	3456	0	0.991	1	9	33	9	24	0			0.510	24 par.m. (24-1v)
	NEG	18	1	2723	11	8	3	718872	9504	0	0.987	1	0	36	0	36	0			0.410	36 par.m. (36-1v)
Accumulated M 1000				2723				4378584	45856				43	170	43			1	0.253		

Type: Type of pattern graph sample, positive or negative.
sample: sample number.
ID test: identification of positive and negative tests for each model graph.
V(M): number of vertices in model graph.
V(P): number of vertices in pattern graph.
mo(M): maximum out-degree for vertices from model graph.
mo(P): maximum out-degree for vertices from pattern graph.
m/effort(*tp+fn*); *(n-c)*/effort(*fp*); *(m-c)*/effort(*fn*); match accuracy (*mA*); accuracy (A); recall (R) and precision (P) are explained with Equation 8.2.

implemented in the `MatchPattern` function and generates a .dot file to visualise the results in GraphViz. See Appendix D for details of these functions. The experiments were run on a Intel machine 2 GHz and 4GB RAM on WinVista.

Figures 8.2, 8.5, 8.6 were derived based on the data from Table 8.1.

Figure 8.2 shows the sampled and accumulated efforts for matching patterns in the directed and undirected[1] versions of target graphs. Efforts are associated to the number of operations that an end user has to perform to identify a given pattern manually (m) and to the number of operations required to transform a match result (obtained automatically by using the pattern matching algorithm) into the *intended* matching result ($(n-c)+(m-c)$) (i.e., adding missing vertices and deleting erroneously matched vertices from the automatically generated result). As explained previously, the *intended* results are those belonging to the class of exact and complete pattern instances, but where surjection is not allowed[2]. For example, sample 7 in Table 8.1 has one false positive, which corresponds to a vertex that illustrates the surjective condition. Figure 8.3 illustrates the latter. In the figure, the result of the matching process in the model graph containing 50 vertices and using the positive pattern sample (S2 in Figure 8.4 and ID test = 2 in Table 8.3) are shown. The target graph shown in Figure 8.3 has two instances of S2: a complete instance and a partial instance. The partial instance consists of a single vertex labelled with a "97"[3] and the complete instance has a vertex labelled with a "174" connected to two vertices labelled with a "97". These two latter vertices are mapped to the single vertex labelled with a "97" in S2 (Figure 8.4), which illustrates the surjection condition. This is also indicated in the eighth row of Table 8.1, where one of the two vertices labelled with "97" is considered a false positive ($f_p = 1$). Note that if surjection is allowed, the algorithm is able to find instances where such a condition holds, which can be frequent and desirable to detect.

Figures 8.5 and 8.6 show values of *match accuracy* (mA - Equation 8.2) for exact matching of positive and negative samples in two cases, when (1) surjection is allowed and complete and partial instances are part of the valid results (true positives); and when (2) only exact and complete pattern instances and no-surjection are allowed. For the latter condition (2), values for the *accuracy* measure (A in Equation 8.1) are also shown. For condition (1), the accuracy value is equivalent to the ideal value i.e., an end user does no need to transform the result obtained automatically with the matching algorithm to obtain the intended result because all the results automatically obtained are 100% accurate. Note that matching of types and attributes is

[1]The undirected versions of graph consider the arcs from the directed versions as vertices. This allows the algorithm to detect information associated with arcs, such as direction.

[2]This condition allows that more than one model graph vertex can be mapped to a single pattern graph vertex (pattern role). See more details in Section 5.2.1.

[3]Note that the vertices labelled with a "97" have also a thicker red line framing them.

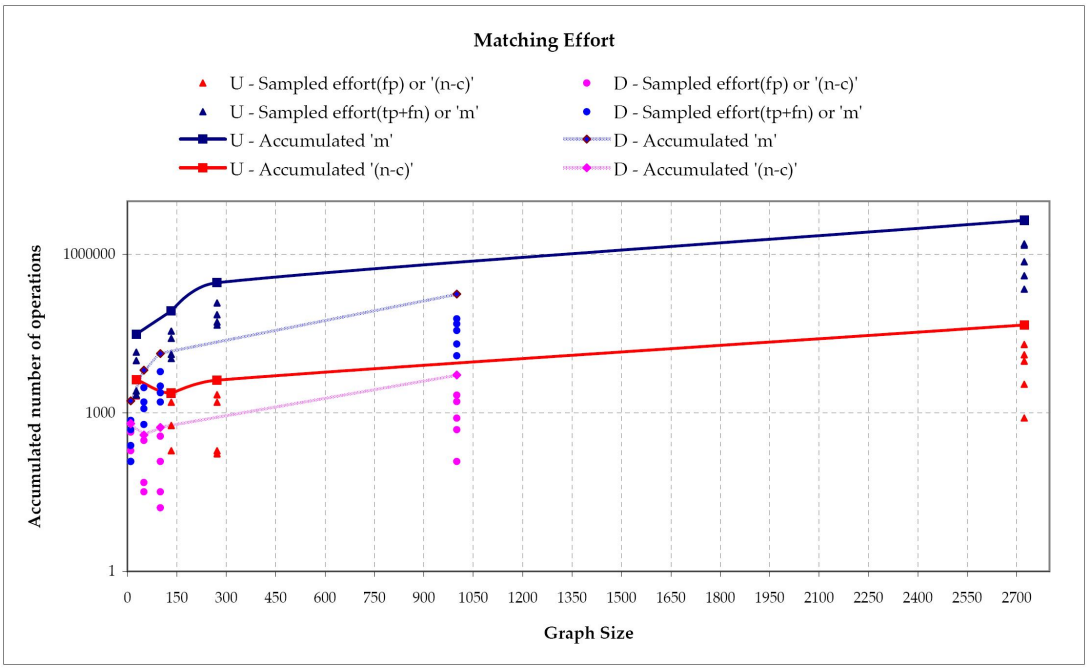


Figure 8.2: Estimation of the accumulated effort to match patterns in directed and undirected versions of target graphs.

not considered here, only structural matching. Accuracy of the algorithm allowing partial matches and surjection, but with complex types and attributes (for example described in natural language) will perform according to the type and attribute matching results. Type and attribute matching here was simplified to matching of simple labels (with a 100% accuracy) in order to isolate results of the structural matching process.

The results of calculating accuracy for matching of positive and negative samples were separated to illustrate the two different trends for these pattern sample conditions. The effort of modifying (if needed) the results of the algorithm for negative samples versus the effort of verifying that the negative sample is not in the graph decreases with the size of the pattern. In small graphs this can be easily done by simple visual inspection. In larger graphs the inspection process requires many more operations than correcting the automated result. Note that corrections are only needed if vertices associated to a surjection condition and partial matches are considered as false positives, otherwise the algorithm performs with the ideal value of match accuracy (mA).

In the case of the accuracy measure (A), for positive and negative samples and when condition (2) holds (i.e., surjection and partial matches are not allowed), the trend of A is opposite to the trend of mA. The larger the graph, the less accurate match results are. This situation can be associated to the probability of a label to ap-

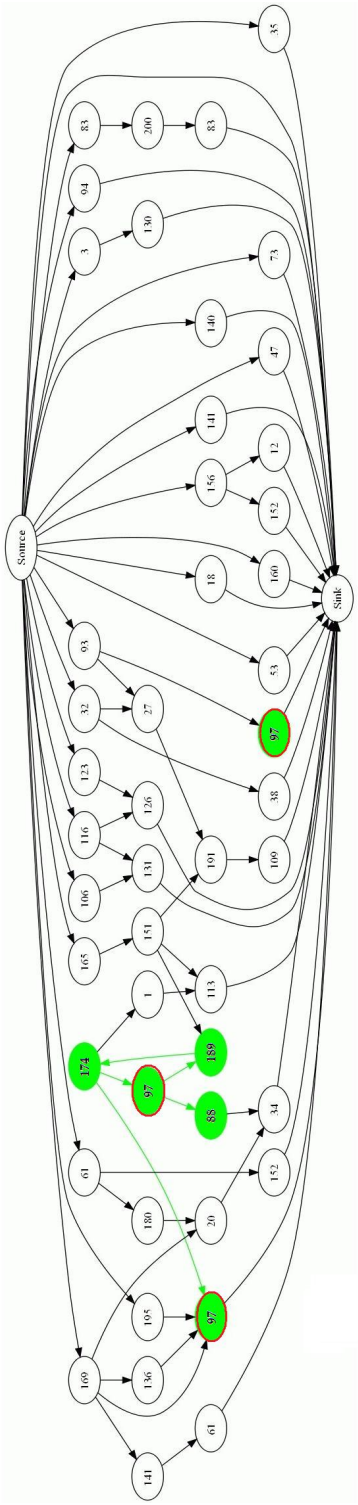


Figure 8.3: Match result for graph M50 and positive pattern sample S2.

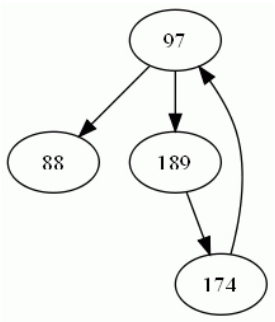


Figure 8.4: Second positive pattern sample (S2).

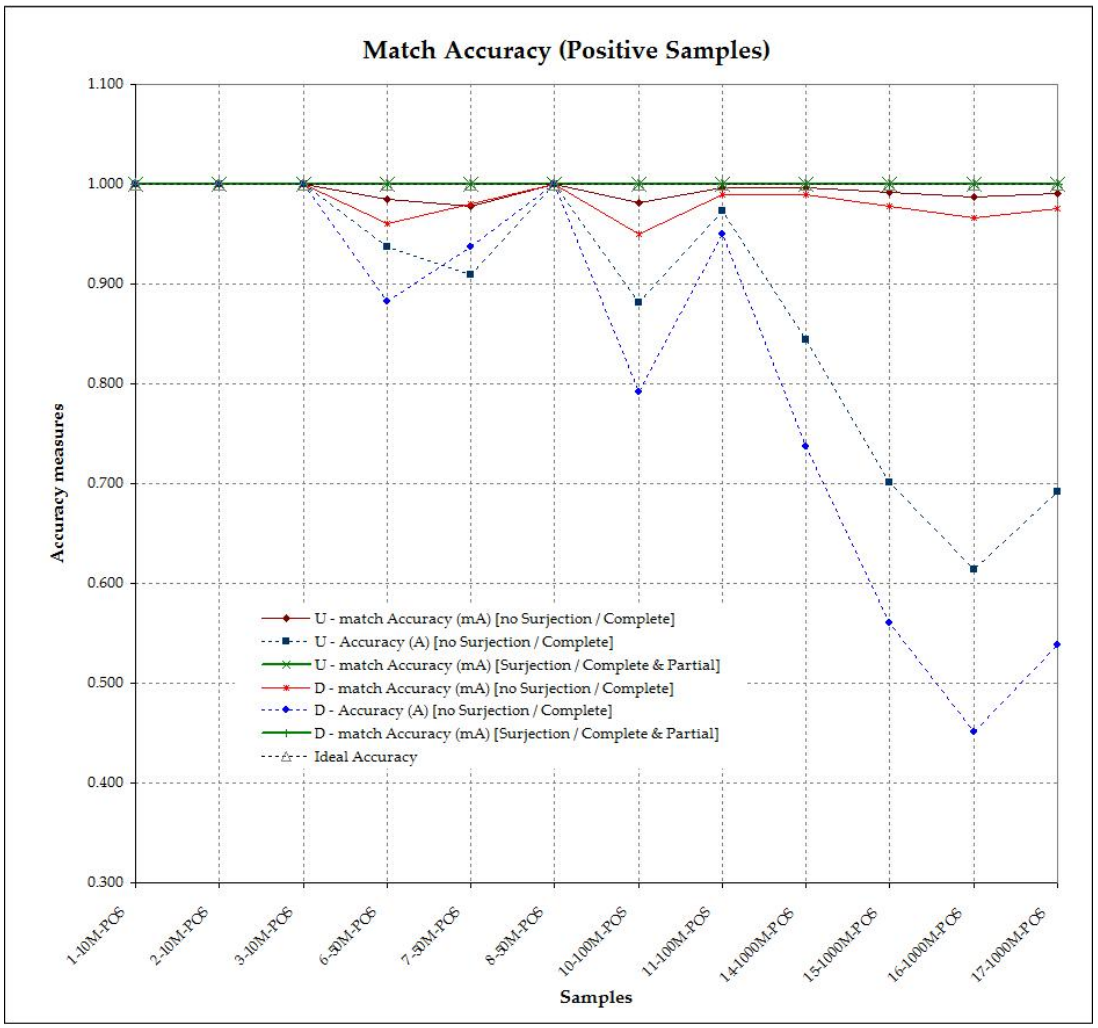


Figure 8.5: Estimation of match accuracy and accuracy for exact pattern matching of positive samples.

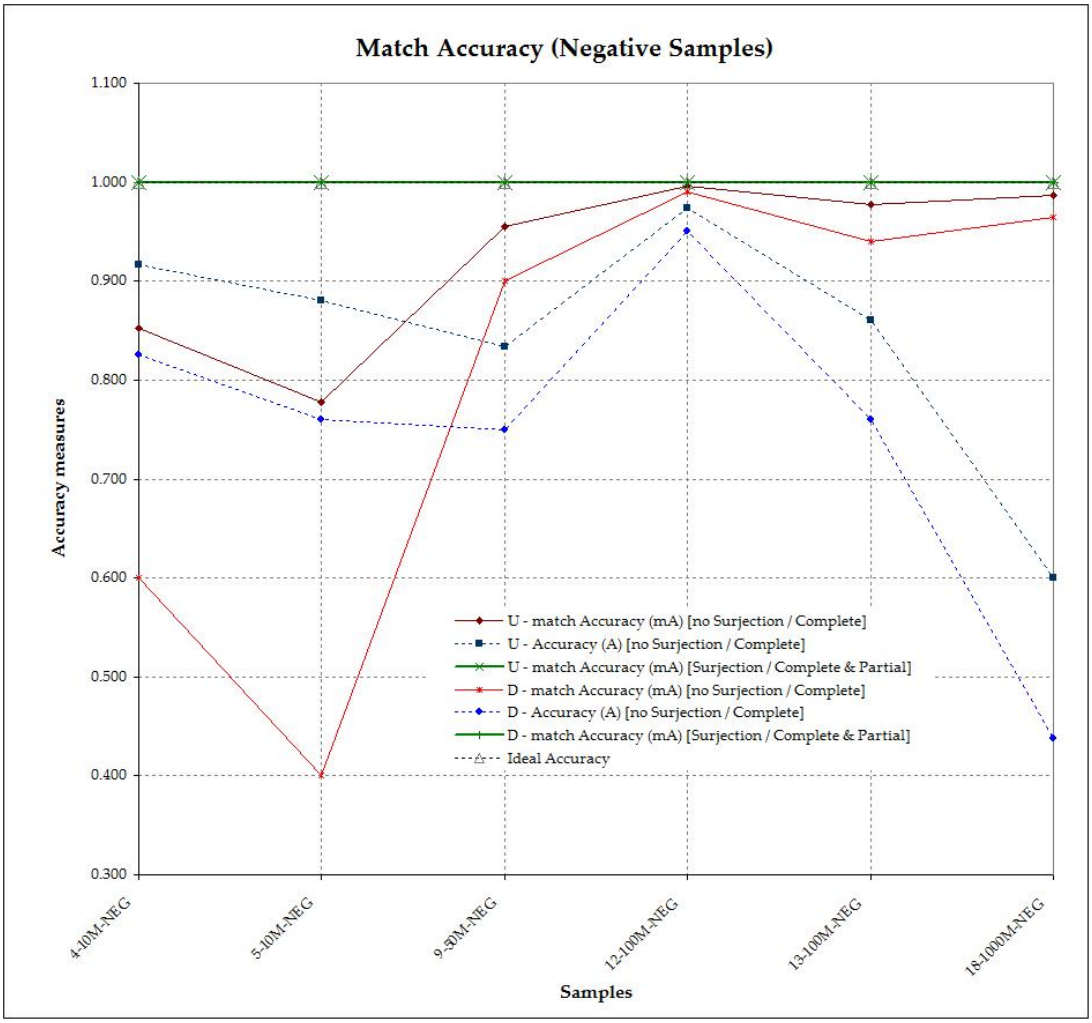


Figure 8.6: Estimation of match accuracy and accuracy for exact pattern matching of negative samples.

pear simultaneously several times in the model graph. Partial matches of one vertex size are more likely to occur in larger graphs, and similar – with less probability – for partial matches of two or more vertices. For instance, for a graph whose size is 10 vertices and there are 200 possible labels the probability of a specific label to be repeated is less than 0.0015. Similar, for graphs of 50, 100 and 1000 vertices and the same 200 possible labels, the probability of having two vertices with the specific label varies (considering the cumulative binomial probability) from 0.002 to 0.014 and to 0.875, respectively. These results indicate that for process models containing many similar elements and only exact and complete matches are required, accuracy A will decrease with the size of the graph, however match accuracy mA will increase.

8.4 Experiments - Processing Time of Pattern Matching

Processing time of the exact and partial matching algorithms is evaluated in a set of experiments considering seven specific patterns over arbitrary random graphs with approximate sizes of 60; 450; 1300; 1800; 3200 and 5000 vertices. The experiments were run on a Intel machine 2GHz and 2GB RAM on WinXP-SP3. For patterns and random graphs three different types of labels are considered, A, B or C. Pattern structures involve four close-walks of two, three, four and six vertices; two line-like patterns of three and four vertices and a star-like pattern of four vertices (Figure 8.7). Line-like structures represent typical sequences of activities in processes. Star-like structures represent points of decision or where the flow of information needs to be split. Close-walks represent cycles or iterations in a process. This type of structure is known to be more complicated to detect in the case of matching approaches considering the possible states of a process. The literature review section (Section 2) provides more details.

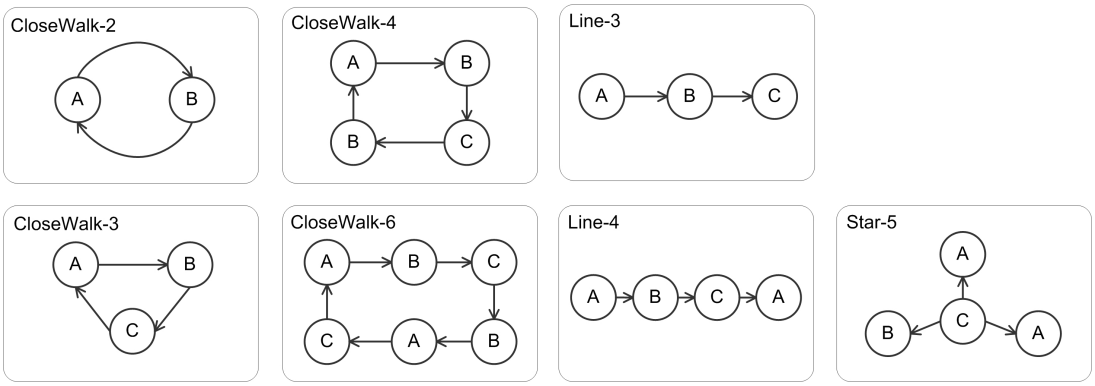


Figure 8.7: Pattern structures used in the experiment.

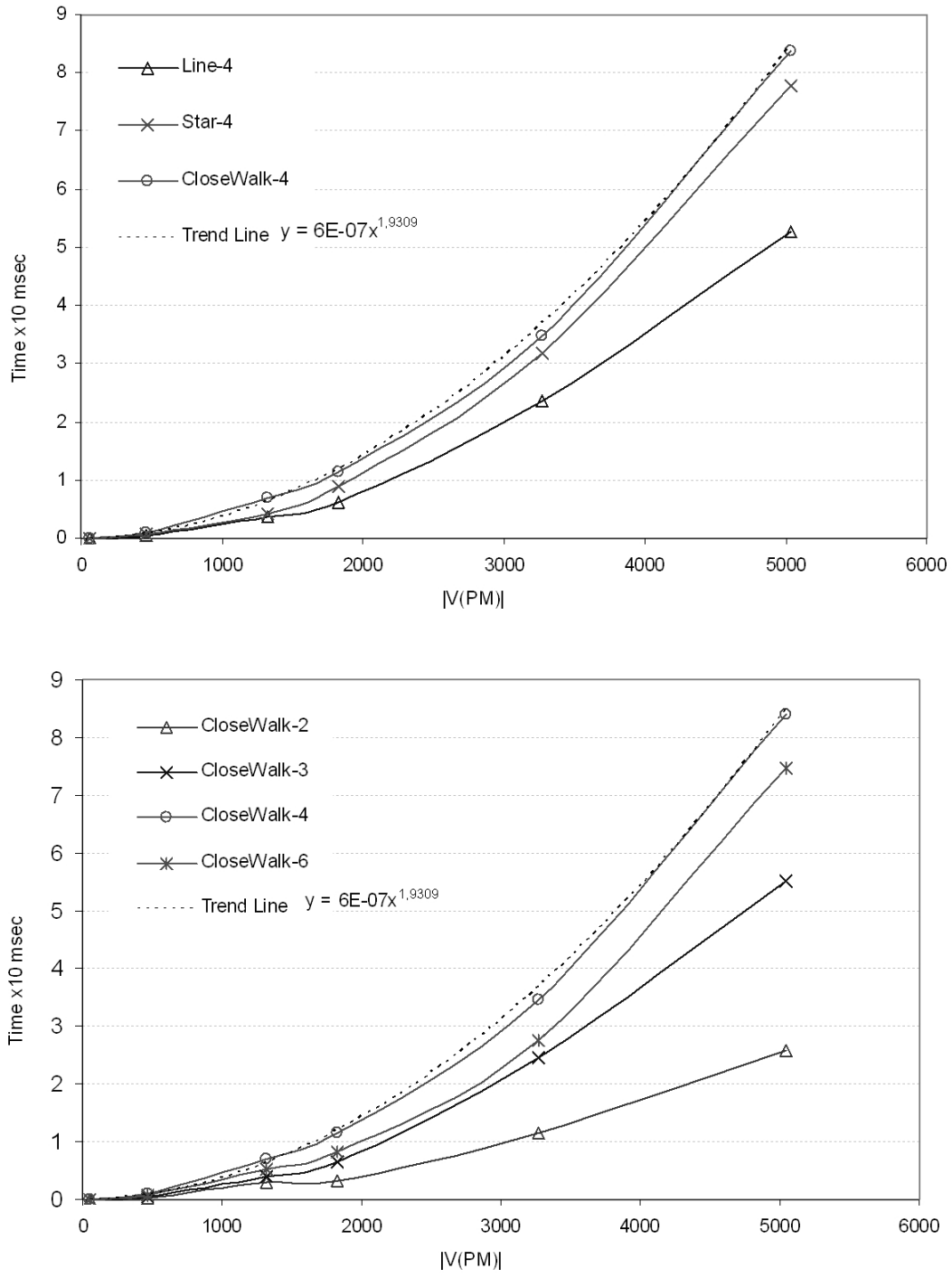


Figure 8.8: Average response time of exact - complete and partial - matching on arbitrary random graphs for different pattern structures (top) and different sizes of a pattern with close-walk structure (bottom).

Figure 8.8 (top) shows the average response time of matching algorithm when matching three patterns with different structures and the same number of vertices. The line-like pattern requires less time in comparison to the star-like and close-walk patterns. It indicates that the structure of matched patterns influences the response time. Figure 8.8 (bottom) shows the average response time of the matching algorithm for patterns with the same structure (close-walk in this case) and different number of vertices. The number of vertices in the pattern also influences the time response. In order to visualise the time response trend more clearly, the time required by the algorithm to compute a solution was divided by the ratio between the number of vertices in the random graph (target graph) and the number of vertices in the pattern. Figure 8.9 (left side) illustrates the trend of the normalised response time for all different patterns considered in the experiment. The right side of Figure 8.9 illustrates the trend of the normalised time response for two patterns with the same structure (close-walk) and different number of vertices (two and six).

The trend lines in Figure 8.8 indicate that the time to solve the problem increases quadratically with the number of vertices in the target graph. The constant 6^{-7} suggest advantageous performance characteristics regarding the response time of the algorithm for small and medium size graphs. The trends in Figure 8.8 indicate that the processing time increases with the size of the pattern and close-walk (or cycles) can be harder to compute. Also, for a same structure of a pattern and target model, the processing time increases linearly with the size of the pattern.

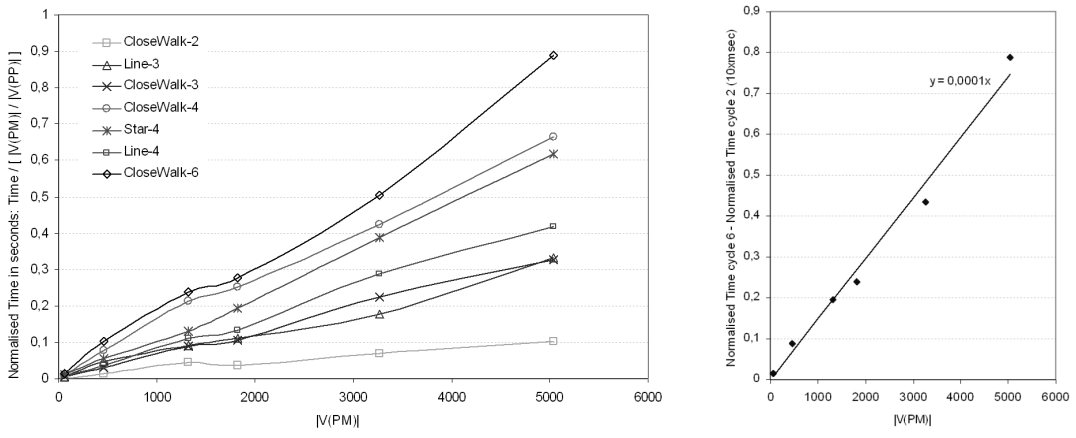


Figure 8.9: Trend of the normalised response time of the matching algorithm on arbitrary random graphs and patterns with different structures.

8.5 Case - Adding Type and Attribute Vertex Matching

Previous sections showed results of the pattern matching process focusing on structural aspects. Matching at vertex label was simplified to a syntactic matching between labels consisting of simple strings (between one and three characters). Actual graph-based process-centric models can contain rich vertex descriptions involving complex vertex types and attributes. Vertex matching beyond syntactic matching can involve a semantic comparison of vertices descriptions. Section 5.5 explains the followed approach for semantic vertex matching. This section demonstrates with a case study how the approach for semantic vertex matching can be applied. The case presented here is based on a publicly available case study provided in the 2008 Workshop on Service- and Process-Oriented Software Engineering[4]. Statistically significant data would require several cases, which were not publicly available at the time of this work to the knowledge of the author. As mentioned previously, a comprehensive evaluation can be considered as part of the future work. A prerequisite would be the creation or adoption of a corpus containing numerous process models and pattern graphs with rich vertex descriptions.

The NRA case. The National Revenue Agency (NRA) is a governmental revenue collection agency which has grown significantly in size and complexity. During its growth, the agency has faced many operational challenges that have triggered rationalization efforts to standardise on emergent process-centric best practices and to reduce operational complexities. The agency has decided to initiate a project to support the spreading and implementation of best practices across the institution. Process-centric best practices are documented as process patterns and they defined an ideal case for processes implementation. Process patterns would lead the definition of new reusable software services. If possible, services would be implemented by exposing functionality of existing software support. The identification of process pattern instances in the actual NRA's processes and their associated legacy applications is the starting point of the rationalization effort. The effort attempts to eliminate redundant legacy applications and to enable best practices automation through software services implementation.

Figure 8.10 illustrates the target processes from NRA. A referential letter u and number next to each element in the process are used to facilitate the explanations. Figure 8.11 illustrates an example of process pattern configurations capturing recommended best practices. They involve the *Validate Form*, *Process Financial Form* and *Process Non-Financial Form* processes. A letter v and number next to each process pattern role is used as a reference throughout the case. The process model and pattern

[4]http://www.dsl.uow.edu.au/sopose/content/files/main/SOPOSE-CaseStudy.pdf.

configurations are modelled using the BPMN 1.1 notation [OMG 2008b].

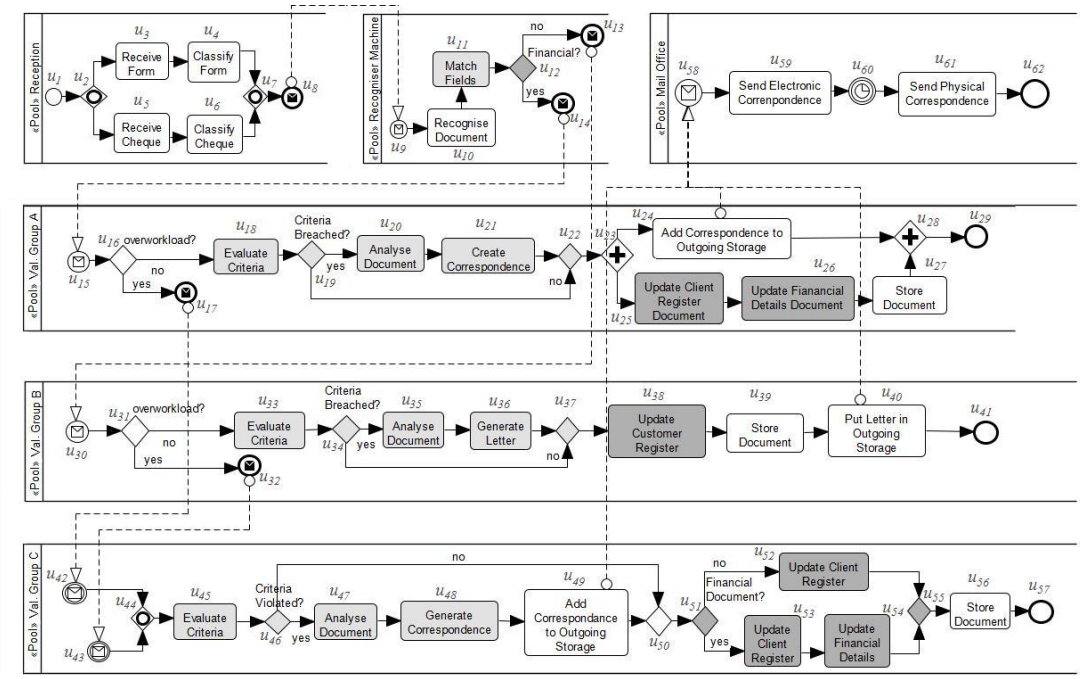


Figure 8.10: NRA process model.

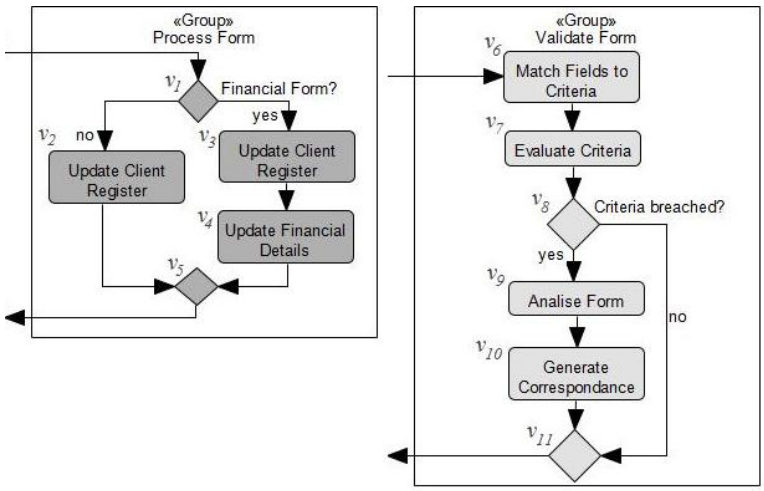


Figure 8.11: Best practices documentation as process pattern configurations.

The pattern matching task involves essentially two stages, one focused on vertex matching and the other on structural matching. The rest of the section is mainly focused on vertex matching. As described in Section 5.5, the vertex matching stage checks if the types of a pattern graph vertex v and a model graph vertex u are equal or the type of u subsumes the type of v - assuming the attributed type graphs of

pattern and model graphs are the same - and if common attributes describing the two vertices are equal or *similar* within a specified similarity threshold. Attribute values would not necessarily satisfy a subsumption relation and they might not always be hierarchically organised.

In the NRA case, a first step would reduce the model graph to vertices whose types have a subsumer type among the types of the pattern graph vertices. In Figure 8.10 these are vertices of activity and decision types. Subsequently, similarity between every two graph vertices u and v (with u a model graph vertex, and v a pattern graph vertex) is calculated by comparing their attribute vectors $\vec{u}$ and $\vec{v}$. The comparison only concerns the attributes describing v. Other attributes in $\vec{u}$ are deliberately omitted since they do not concern the pattern. The choice of the attributes describing a pattern is made by considering both its intended application and also its potential users. In this example, existence of a common set of attributes describing vertices from the pattern and the model graph is assumed. Similarity between the attribute vectors $\vec{u}$ and $\vec{v}$ is calculated based on Equation 5.1 (Section 5.5), which is re-written below.

$$sim(\vec{u},\vec{v}) = 1 - \left(|\delta_i \cdot dis(\vec{u}_i,\vec{v}_i)|^p\right)^{1/p}, 1 \leq i \leq |\vec{v}| \tag{8.4}$$

$dis(\vec{u}_i,\vec{v}_i)$ is the normalised *dissimilarity* between $\vec{u}$ and $\vec{v}$ in the attribute i and δ_i is a weighting factor to emphasize or deemphasize the *ith* attribute value. For this example attributes are assumed independent of each other. p is chosen equal to 2, defining that vertex similarity would be based on Euclidean distance.

One of the most common attributes for vertices in process graphs is the label, which often is a sentence described in natural language. Matching labels requires a complex calculation involving the semantics associated to concepts referred by words in labels. Since matching labels is one example of a complex case associated to vertex matching, the NRA case focuses on matching labels of vertices. Aggregation of another attributes and types associated to vertices was already explained in Section 5.5.

Matching of labels uses the *sentence similarity* measure described in [Li 2006]. It is sufficient in this context since the elements required to evaluate the measure are dynamically generated using only the information from the words contained in the two labels. The measure considers the semantic similarity between words in the two sentences (labels), which is derived from a Lexical Knowledge Base (LKB) and a corpus, and the word order on the sentence meaning. LKBs are frequently organised as a hierarchy of words defining concepts (for example, WordNet[5] or other more specific LKBs targeting particular business domains). Semantic similarity between

[5]Available at http://wordnet.princeton.edu/

words is calculated based on the length of the path connecting the words in the hierarchy and their depth in it. By observing the direction (from bottom to top) of the path connecting two words in the hierarchy, it is possible to discriminate between abstraction or refinement of concepts. The latter can be used as indication that a vertex label is an abstraction of another vertex label.

The explanation of vertex label similarity calculation is simplified by avoiding word disambiguation, which requires the analysis of the context where the word appears, abbreviations expansion and acronyms replacement.

The label similarity measure described in Equation 5.2 (Section 5.5) is used to identify vertex labels in the model graph similar to vertex labels in the pattern graph. Label similarity is a weighted measure between the similarity of lexical and word order vectors of two vertices u and v (Equation 5.3 and 5.7). The three metrics mentioned before are re-written here to illustrate how label similarity is calculated between one vertex from the pattern graph and three similar vertices from the model graph.

$$sim_{label}(\ell(u), \ell(v)) = \rho \cdot sim(\vec{w}(u), \vec{w}(v)) + (1 - \rho) \cdot sim(\vec{o}(u), \vec{o}(v)) \tag{8.5}$$

$$sim(\vec{w}(u), \vec{w}(v)) = \frac{\vec{w}(u) \cdot \vec{w}(v)}{\|\vec{w}(u)\| \|\vec{w}(v)\|} \tag{8.6}$$

$$sim(\vec{o}(u), \vec{o}(v)) = 1 - \frac{\|\vec{o}(u) - \vec{o}(v)\|}{\|\vec{o}(u) + \vec{o}(v)\|} \tag{8.7}$$

Consider now the vertex label $\ell(v_3)$: *Update Client Register* from the process pattern configuration in Figure 8.11, and the vertex labels $\ell(u_{25})$: *Update Client Register Document*, $\ell(u_{38})$: *Update Customer Register* and $\ell(u_{52})$: *Update Client Register* from Fig. 8.10. We want to calculate the similarity between $\ell(v_3)$ and the mentioned labels $\ell(u_{25})$, $\ell(u_{38})$ and $\ell(u_{52})$. The associated joint word sets are:
$W_{(v_3,u_{25})} = \{Update\ Client\ Register\ Document\}$,
$W_{(v_3,u_{38})} = \{Update\ Client\ Register\ Customer\}$,
$W_{(v_3,u_{52})} = \{Update\ Client\ Register\}$

The lexical semantic vectors associated to each joint word set are shown below:
$W_{(v_3,u_{25})}$: $\vec{w}(v_3) = [1\ 1\ 1\ 0]$, $\vec{w}(u_{25}) = [1\ 1\ 1\ 1]$
$W_{(v_3,u_{38})}$: $\vec{w}(v_3) = [1\ 1\ 1\ 0.8182]$, $\vec{w}(u_{38}) = [1\ 0.8182\ 1\ 1]$
$W_{(v_3,u_{52})}$: $\vec{w}(v_3) = [1\ 1\ 1]$, $\vec{w}(u_{52}) = [1\ 1\ 1]$
See Section 5.5.4 for a detailed explanation of lexical semantic vector derivation.
The WordNet::Similarity service[6] was used to obtain the path length between the

[6] Available at http://wn-similarity.sourceforge.net/

compared words and depth of the common subsumer, and replaces those in Section 5.7.

Word order vectors were obtained as in the following example. Consider the joint word set $W_{(v_3,u_{38})} = \{Update\ Client\ Register\ Customer\}$ and its associated joint word order vector $\vec{O}_{(v_3,u_{38})} = [1\ 2\ 3\ 4]$. The word order vector for v_3 is $\vec{o}(v_3) = [1\ 2\ 3\ 2]$. The first three entries in $\vec{o}(v_3)$ relate to words in $\ell(v_3)$, the last entry (associated to the word *Customer*) is not in $\ell(v_3)$, but the most similar word is *Client*, and consequently the last entry in $\vec{o}(v_3)$ is the index of *Client* in $\vec{O}$. Analogously for u_{38}, $\vec{o}(u_{38}) = [1\ 4\ 3\ 4]$. After calculating the lexical semantic vector similarities and word order vector similarities, the label similarity between $\ell(v_3)$ from the pattern *Validate Form* in Figure 8.11 and each of the "matched" labels associated with vertices in the process model from Figure 8.10 was calculated according to Equation 5.2. It was considered a $\rho = 0.85$ according to the experimental findings in [Li 2006]. Thus,
$sim_{label}(\ell(v_3), \ell(u_{25})) = 0.8154$,
$sim_{label}(\ell(v_3), \ell(u_{38})) = 0.9523$, and
$sim_{label}(\ell(v_3), \ell(u_{52})) = 1.0000$.

After repeating all previous steps for all vertices in the model and pattern graphs, successful individual vertex matches start an expansion stage in a breadth first search manner until the final (complete or partial) pattern instances are identified in the model graph. Elements in the model from Figure 8.11 highlighted in light-grey and dark-grey are the initial matched vertices. The same case explained here is also described in [Gacitua-Decar 2009a], for more details see the previous reference.

8.6 Experiments - Frequent Subgraph Discovery

Pattern matching aims to identify instances of a known pattern in a model. Graph-based pattern discovery aims to find frequent subgraphs within a given graph. For process-centric models, Chapter 6 proposes a frequent subgraph discovery algorithm based on the graph pattern matching algorithms in Chapter 5. This section is divided into two. First, an example to explain the type of results generated by the algorithm is given and second, a description of an experiment exploring the influence of the number of available labels for the frequent subgraph discovery task is provided. In the experiment, random labels from a fixed number of different labels can be assigned to vertices in the target processes.

8.6.1 Case to explain the algorithm's results

Consider the process model and associated graph in Figure 8.12, the target graph of the pattern discovery algorithm. The model contains two maximum substructures

repeated more than two times. The frequency of the substructures indicates an opportunity for potential reuse (at process level or the underlying software support). For a graph of this size, finding substructures by simple inspection of the model could be less complicated, but for models with numerous elements and hierarchically arranged substructures, automated support could be required.

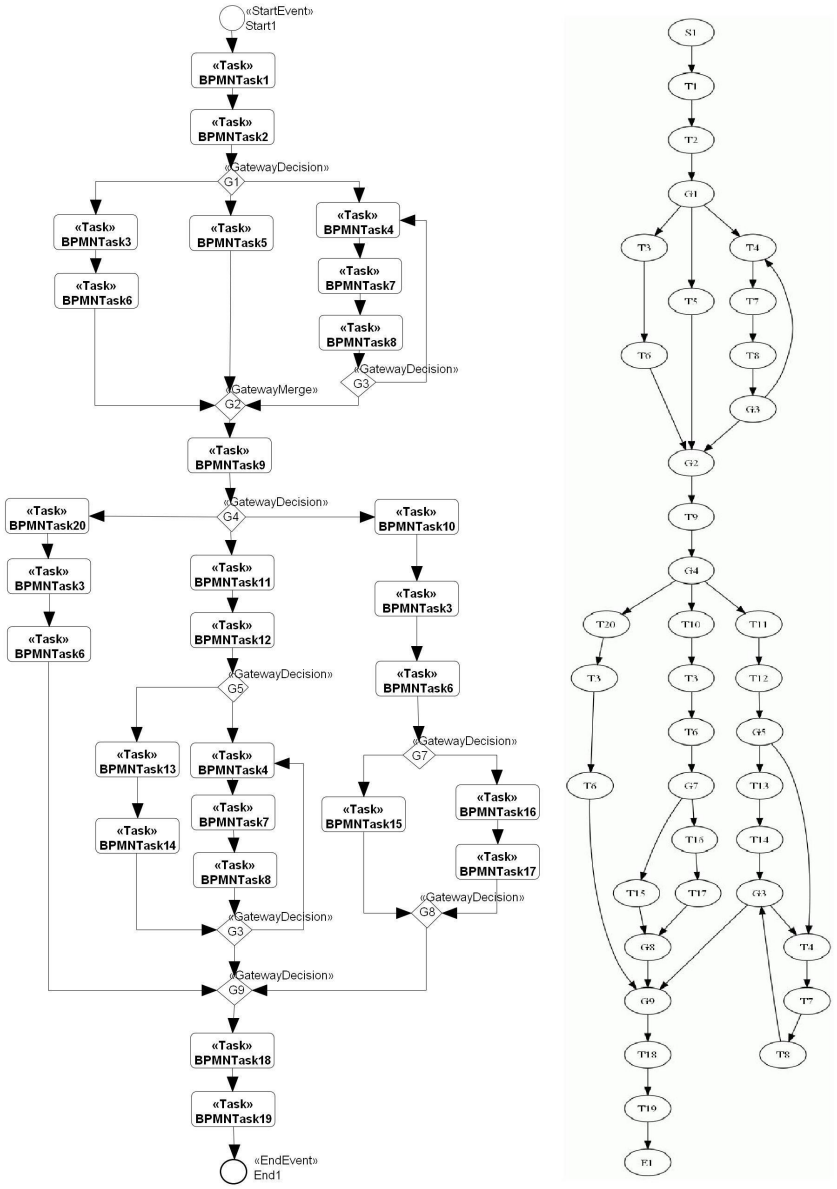


Figure 8.12: Process model and associated graph (process) model.

The proposed algorithm, implemented in function `FindPatterns4` in Appendix D, automates the task by taking the model (M), generating temporal (potential) patterns based on the expansion of each vertex in M, and matching them with the rest of

the model. The function `testDiscovery1` in Appendix D (that uses `FindPatterns4`) allows us to discover frequent patterns and to visualise the results in a graph model. The results are two matrices – `ScoreFoundPatterns` (*SFPM*), *FreqMatrix* (*FM*) – and two graphical representations of the model - a directed and an undirected version - with the discovered patterns highlighted.

The rows in both matrices (*SFPM* and *FM*) indicate the indexes of vertices in the undirected version of the graph model. Columns in *SFPM* indicate size scores of discovered patterns. A particular entry in the matrix would indicate the size score of a vertex u in M for a particular pattern size. This size score refers to the number of vertices of the discovered pattern where u belongs. If u is not an element of a discovered pattern, then the score equals zero. Two input parameters modulate the search of frequent subgraphs in M. In function `FindPatterns4`, they are indicated as `stepsPattern` and `Threshold`. The former defines the size of the discovered pattern by indicating the amount of times a temporal pattern is expanded to initiate a search across the model graph. The latter indicates a threshold value for the vertex matching step. This is especially relevant in semantic vertex matching. For tests ran in this example, the threshold was set to one (indicating only exact vertex matches are allowed) and the `stepsPattern` variable was varied from one to four – i.e., the tests would indicate if patterns of size between $1 \cdot min[In/Out - degree(u), u \in M]$ and $1 \cdot max[In/Out - degree(u)]$ to $4 \cdot min[In/Out - degree(u), u \in M]$ and $4 \cdot max[In/Out - degree(u)]$ would exist in M. Columns in *FM* indicate the relative frequency of different size patterns created from each vertex in the model. A particular entry of the matrix, associated to a vertex u in the model, refers to how many times a pattern originated in u and with a size relative to the number of expansion steps (regulated with `stepsPattern`) occurs in the model. For instance, for the entry indicating a frequency equal to two, for a vertex u in a specific row of matrix *FM* and fourth column, it would mean that a pattern created from u, which is expanded three times (fourth column), is two times in the model.

Figure 8.13 shows the results indicated in *FM* for subgraphs (potential patterns) centered on each model graph vertex. For zero expansion steps (i.e., a potential pattern would contain only one vertex) all vertices appear in *FM* with at least a value equal to one. For four expansion steps, there are no vertices that can derive a pattern which would appear more than two times. The closest are the vertices from the undirected version of the model representing the edges connecting $G4$ to $T20$ and $G4$ to $T10$, which are more or less in the graph centre. Given the size of the model and the restriction of four expansion steps, patterns which are created from the graph centre are more likely to be at least two times in the graph.

Figure 8.14 shows only those vertices that can generate patterns occurring more

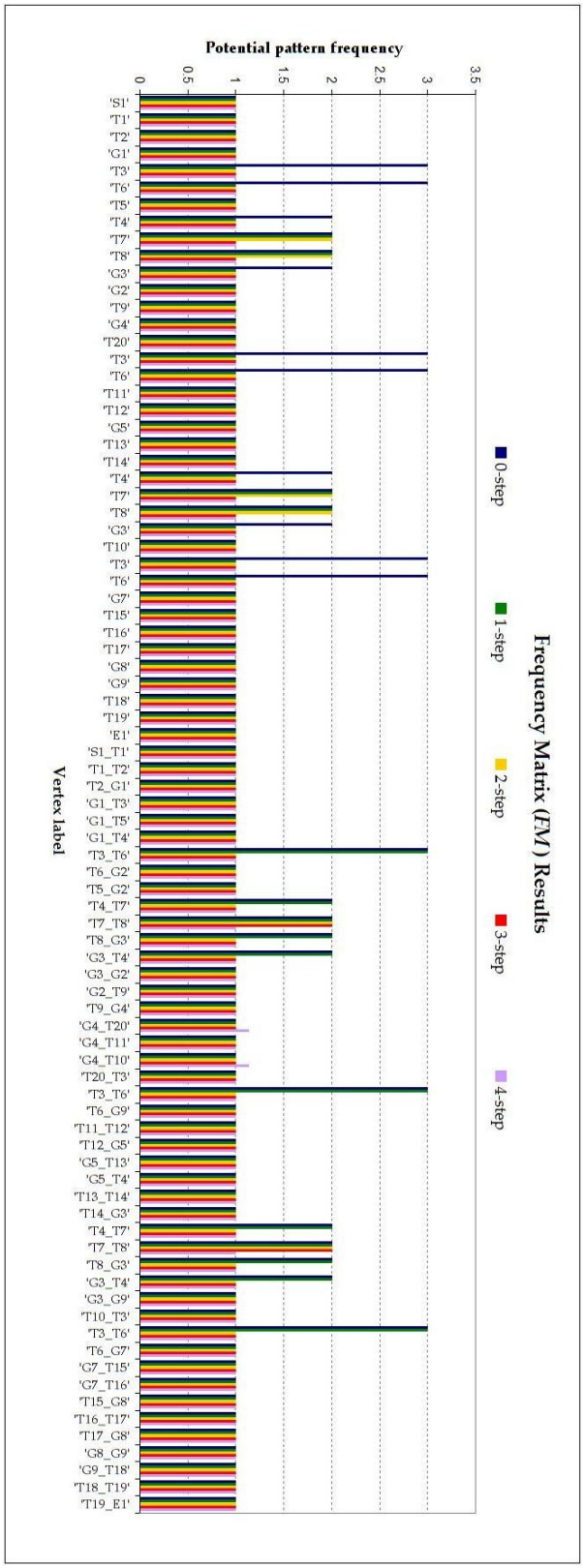


Figure 8.13: Frequency of subgraphs (potential patterns) centered in each model graph vertex.

than two times in the model and whose size would be defined by one, two or three steps of expansion. The results indicate that there are three instances of a subgraph generated by expanding in one step a vertex from the undirected version of the model representing the edge connecting $T3$ to $T6$ (named here *Pattern A*), and two instances of a subgraph generated by expanding in three steps a vertex from the undirected version of the model representing the edge connecting $T7$ to $T8$ (named here *Pattern B*). The graphic also indicates that subgraphs of the *Pattern B* centred in $T7$, $T8$ created with two expansion steps would occur two times in the model. The same holds for subgraphs of the *Pattern B* created with one expansion step and centered in the edges connecting $T4$ to $T7$, $G3$ to $T4$ and $T8$ to $G3$.

Results of the pattern discovery algorithm for one expansion step and frequency of discovered patterns greater than two is visualised in Figure 8.15. If the discovered patterns would have been known beforehand, the pattern matching algorithm explained in the previous sections could be used to match them automatically. Results of matching four given patterns (see Figure 8.16), including the discovered *Pattern A* and *B* are shown in Figure 8.17.

8.6.2 Effects of Varying the Size of the Vertex Descriptors' Set

The discovery of frequently occurring substructures on large scale graphs representing process-centric models can be used as an instrument to guide the definition of reusable process sections as early designs of process-centric services. Explicit (trace) links between discovered frequent process patterns and existing software components can provide information of potential reuse at software levels. Process-centric services, as architecture elements between process and software layers can be designed taking the guidelines of the discovered (frequent) process patterns to enhance services reuse. For example, see Figure C.18 in Appendix C illustrating a possible implementation of the *Loan to Client* pattern (from the *LM* case study in the previous section) as a service that is based on functionality available from existing applications.

One of the aspects to consider when designing services is the favorable (or unfavorable) characteristics they have when they are composed with other services. For instance, the influence of a single service on the performance of a service composition. Composite services involving numerous granular services tend to have a worse performance than those composed of less and more granular services. If frequent process patterns are considered as guidelines to design reusable services, the size and frequency of them are two relevant aspects to take into account. The more frequently a pattern occurs, the more chances it will be considered as a guideline to design a potentially more reusable service. Also, the higher the frequency of the

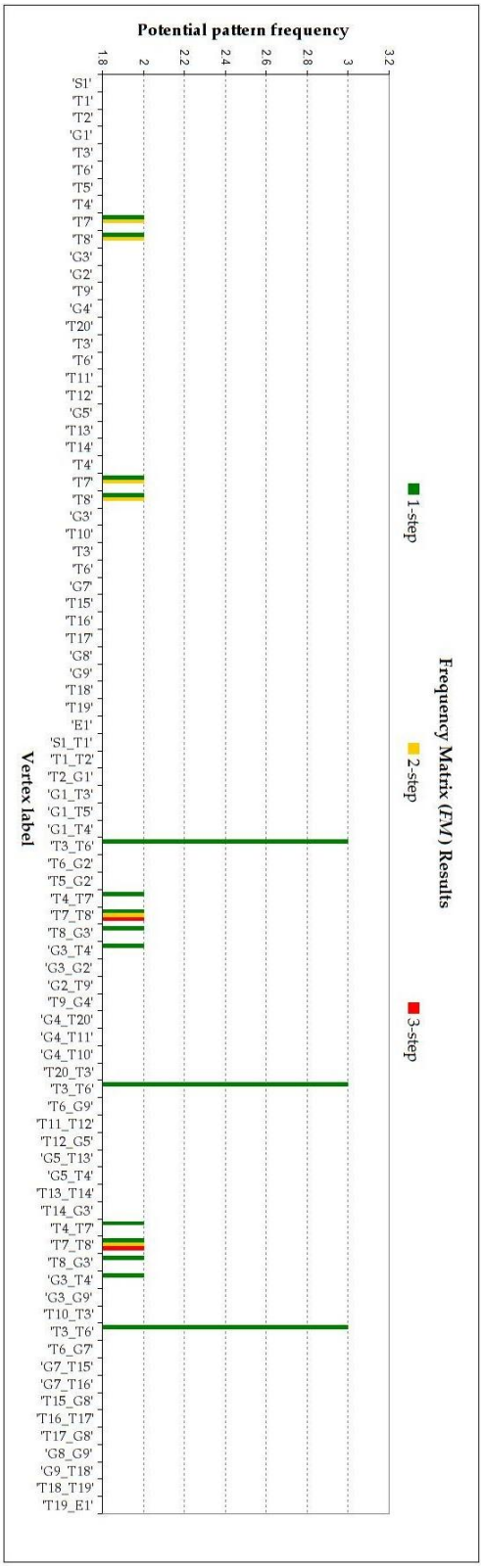


Figure 8.14: Frequency results for subgraphs appearing more than two times in the graph model.

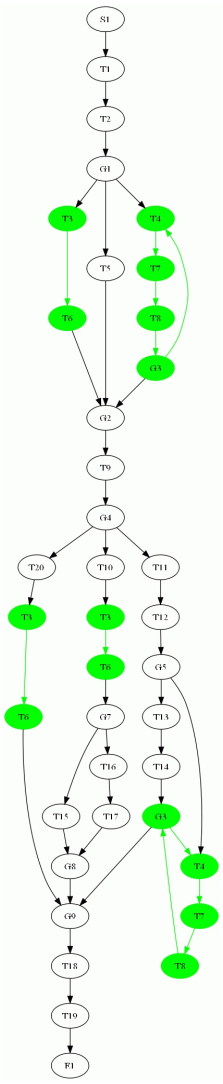


Figure 8.15: Visualisation - in GraphViz, Section 8.7.3 - of pattern discovery algorithm results. One expansion step and frequency greater than two.

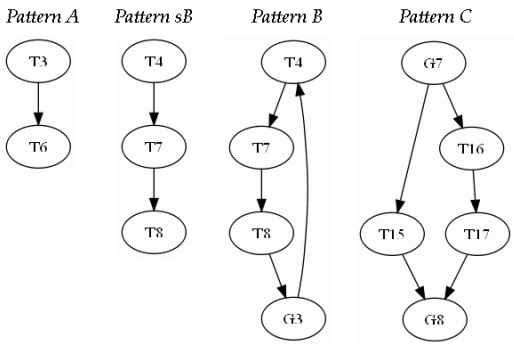


Figure 8.16: Sample patterns, including discovered *Pattern A* and *B*.

	Pattern A	Pattern sB	Pattern B	Pattern C
S1	0	0	0	0
T1	0	0	0	0
T2	0	0	0	0
G1	0	0	0	0
T3	2	0	0	0
T6	2	0	0	0
T5	0	0	0	0
T4	0	3	4	0
T7	0	3	4	0
T8	0	3	4	0
G3	0	3	4	0
G2	0	0	4	0
T9	0	0	0	0
G4	0	0	0	0
T20	0	0	0	0
T3	2	0	0	0
T6	2	0	0	0
T11	0	0	0	0
T12	0	0	0	0
G5	0	0	0	0
T13	0	0	0	0
T14	0	0	0	0
T4	0	3	4	0
T7	0	3	4	0
T8	0	3	4	0
G6	0	3	4	0
T10	0	0	0	0
T3	2	0	0	0
T6	2	0	0	0
G7	0	0	0	5
T15	0	0	0	5
T16	0	0	0	5
T17	0	0	0	5
G8	0	0	0	5
G9	0	0	0	5
T18	0	0	0	0
T19	0	0	0	0
E1	0	0	0	0

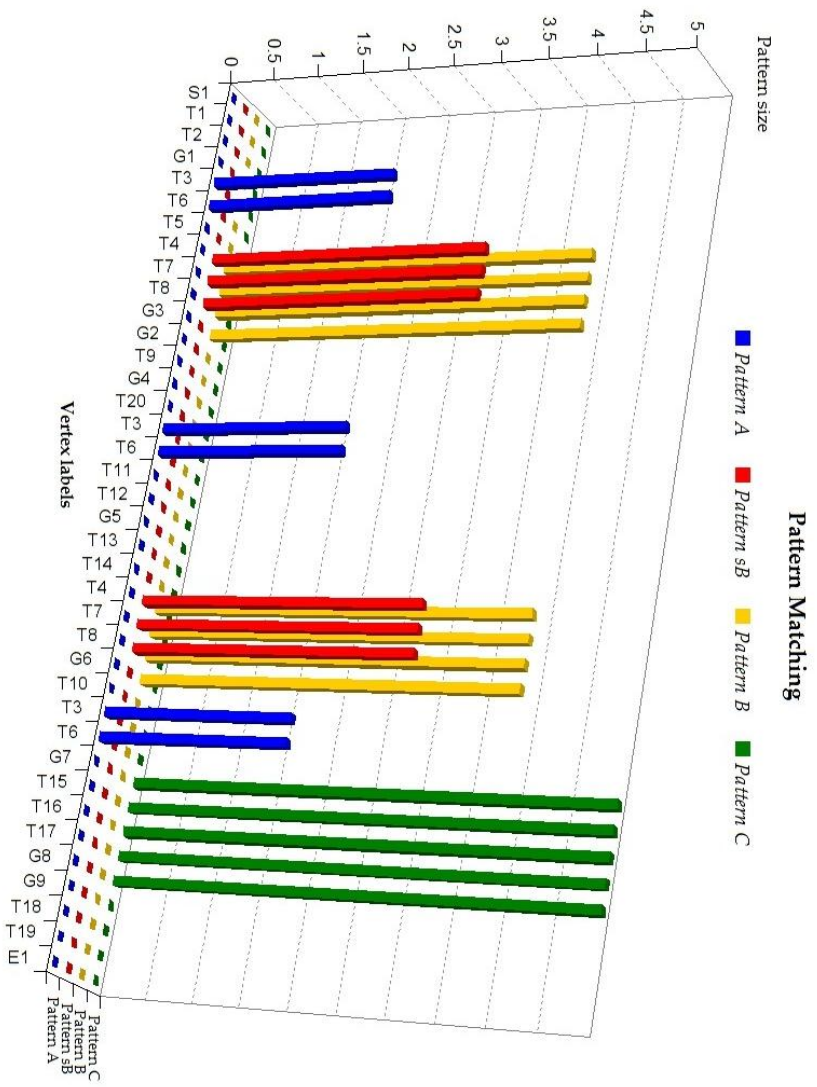


Figure 8.17: Results of matching known patterns, including those discovered.

pattern, the more probable it is that the granularity of the service derived from this pattern would be smaller. When using this strategy to define all services (services based on highly frequent patterns whose size is probably small), it can be expected that the global performance of a service composition involving several of these fine granular services is worse than a composition involving a few coarse grained services. A tradeoff between reusability and performance should be decided by the architect. Thus, s/he can decide the values for the size and frequency parameters of the pattern discovery algorithm.

The following figures show how the frequency of discovered patterns varies with the number of expansion steps in the discovery algorithm and with the label set size. Variations of the number of expansion steps are used to investigate the influence of the patterns size over their frequency. The finding regarding the influence of changing the label set size over the frequency of discovered patterns is used as an example of the effects in the patterns' frequency due to variations in the size of the sets containing vertex types and attributes.

Figures 8.18 and 8.19 show the results in the frequency matrix for each vertex in process graphs of 50 and 100 vertices and with label set sizes varying between 10, 20, 50 and 100 different labels. Only frequencies greater than one are shown. A frequency greater than one indicates that a subgraph derived from expanding a vertex with its neighbours occurs more than one time in the graph model. A frequency equal to two indicates that the subgraph occurs two times. A frequency between one and two, indicates that there is one partial instance of the subgraph occurring somewhere else in the model graph. In general, when the value of the frequency is a fraction, it indicates that the subgraph (potential pattern) has one or more partial instances in the graph model or one (or more) of its instances contain more than one pattern role instance. The latter refers to the surjection condition that has been mentioned several times throughout the chapter. From Figures 8.18 and 8.19, it can be estimated that if the the set of possible labels is smaller, there are more chances of finding more frequent substructures. At the same time, the smaller the size of the subgraphs (i.e., less expansion steps in the algorithm) the more chances for the subgraphs (potential patterns) to appear more frequently in the target graph.

Figures 8.20 and 8.21 show the average frequency and +/- values for a 95% confidence interval of potential patterns in two graphs. One graph has 50 vertices and the other one has 100 vertices. The label set sizes for both graphs vary between 10, 20, 50 and 100 different labels. The figures indicate that patterns with a size greater than one vertex and appearing more than two times in the graph only occur (in *average*) for the target graph having 100 vertices and with the algorithm running with one expansion step. For algorithm running with two expansion steps, patterns appear in the graph (on average) less than two times. Note that these results refer to average

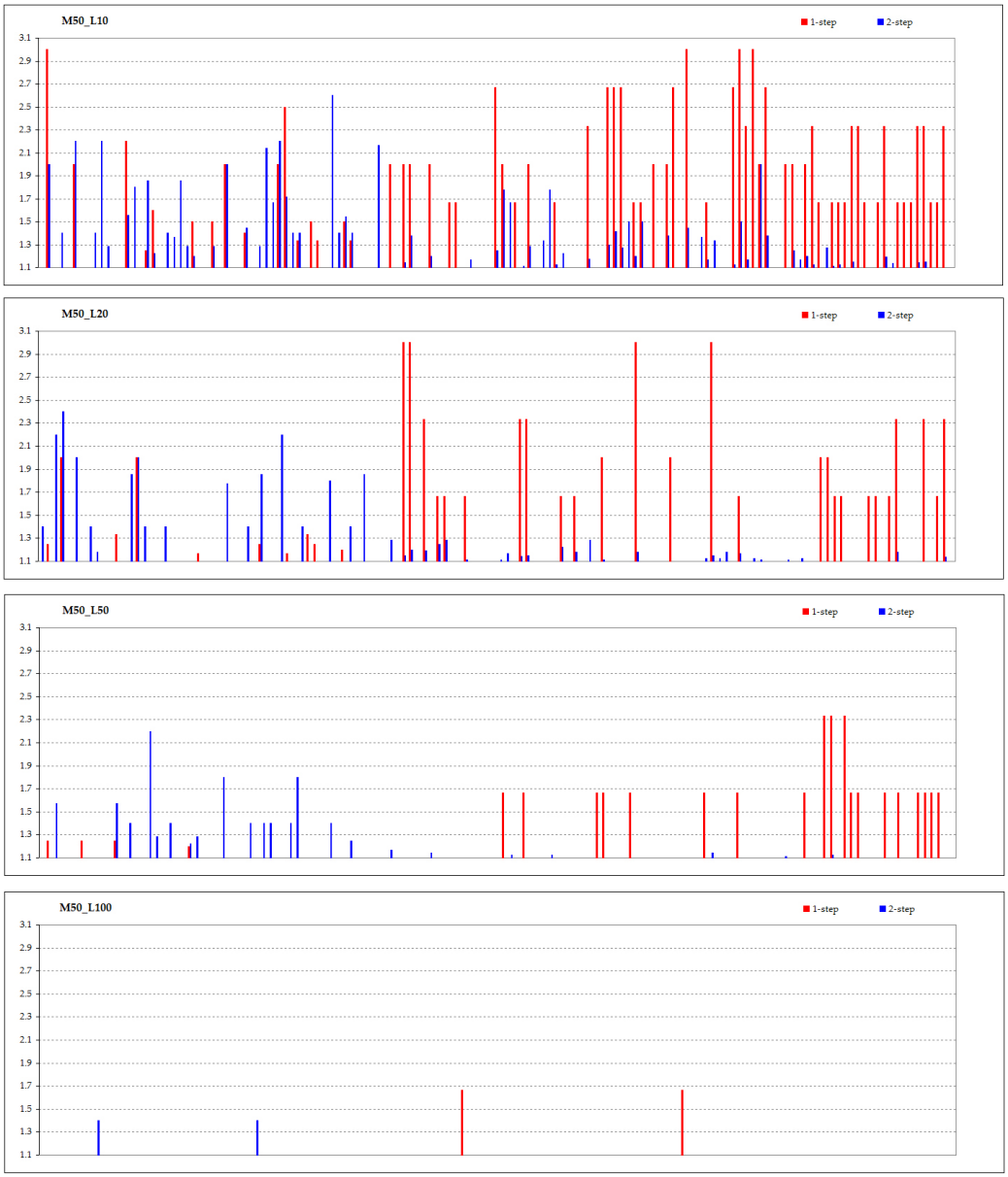


Figure 8.18: Graph M50 - frequency matrix (*FM*) results greater than one, with one/two expansion steps.

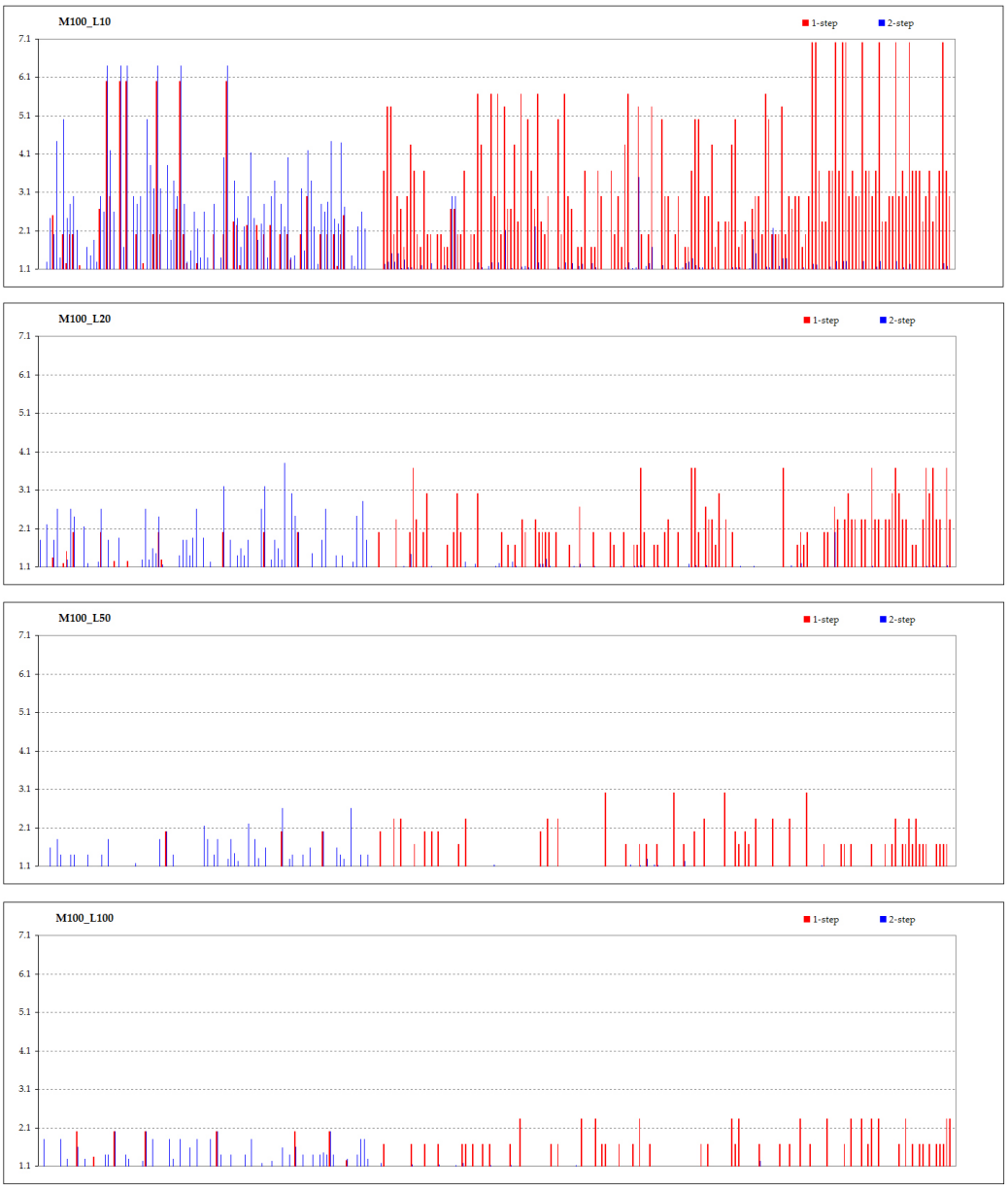


Figure 8.19: Graph M100 - frequency matrix (*FM*) results greater than one, with one/two expansion steps.

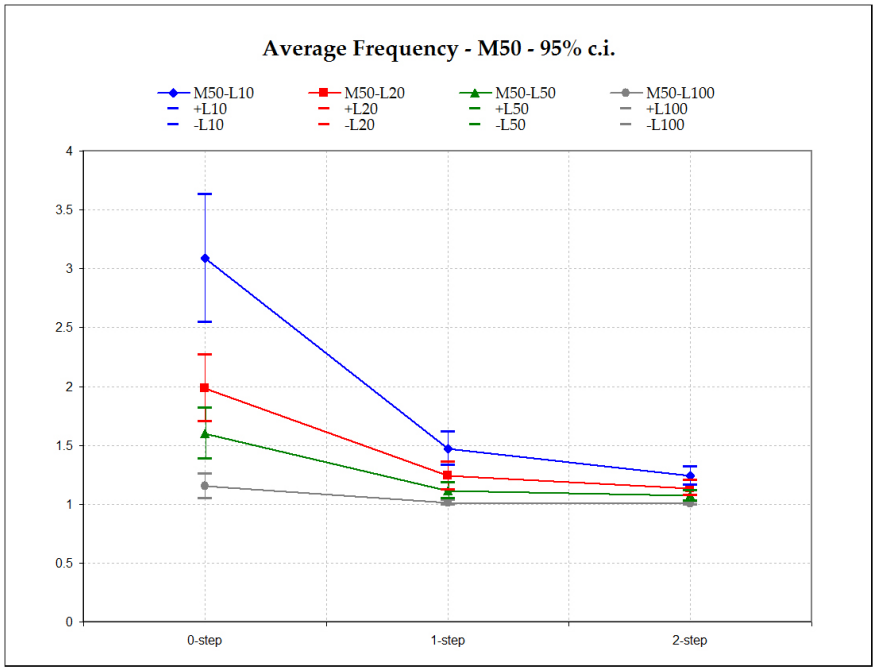


Figure 8.20: Average frequency of potential patterns. Graph size = 50, 95% confidence interval.

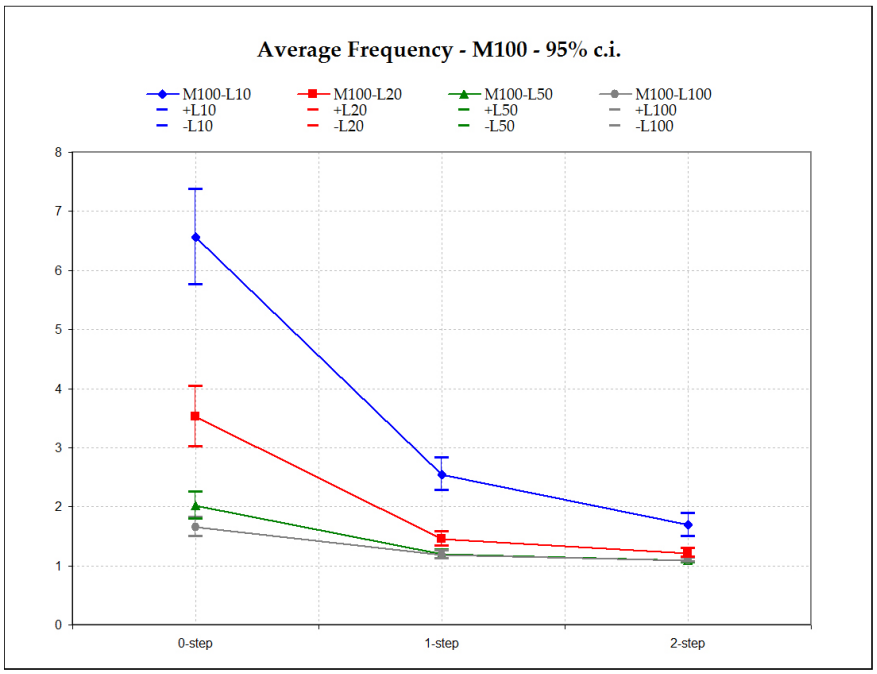


Figure 8.21: Average frequency of potential patterns. Graph size = 100, 95% confidence interval.

values. Absolute values were already shown in Figures 8.18 and 8.19.

8.7 Tool Support

This section focuses on the implementation and experimental set-up of the pattern matching and discovery algorithms. Figure 8.22 illustrates an overview of the software components used to implement and evaluate the proposed techniques. The pattern *matching* and *discovery* techniques are implemented as Matlab functions. Modelling support, model pre-processing steps and *pattern documentation* facilities are implemented as a UML-based profile, facilities in a standard CASE tool and model-to-graph transformation component (see Section 7.3.1 for more details). Graph *generation* is implemented in Matlab functions. Graph *visualisation* support considers an external tool (GraphViz). Elements framed as *extensions* illustrate the capabilities to extend the current tool support. Components framed in dotted lines are concrete examples of new components extending the framework. A lexical semantic network (WordNet) and the word similarity service (Wordnet::Similarity) are external components used during the evaluation.

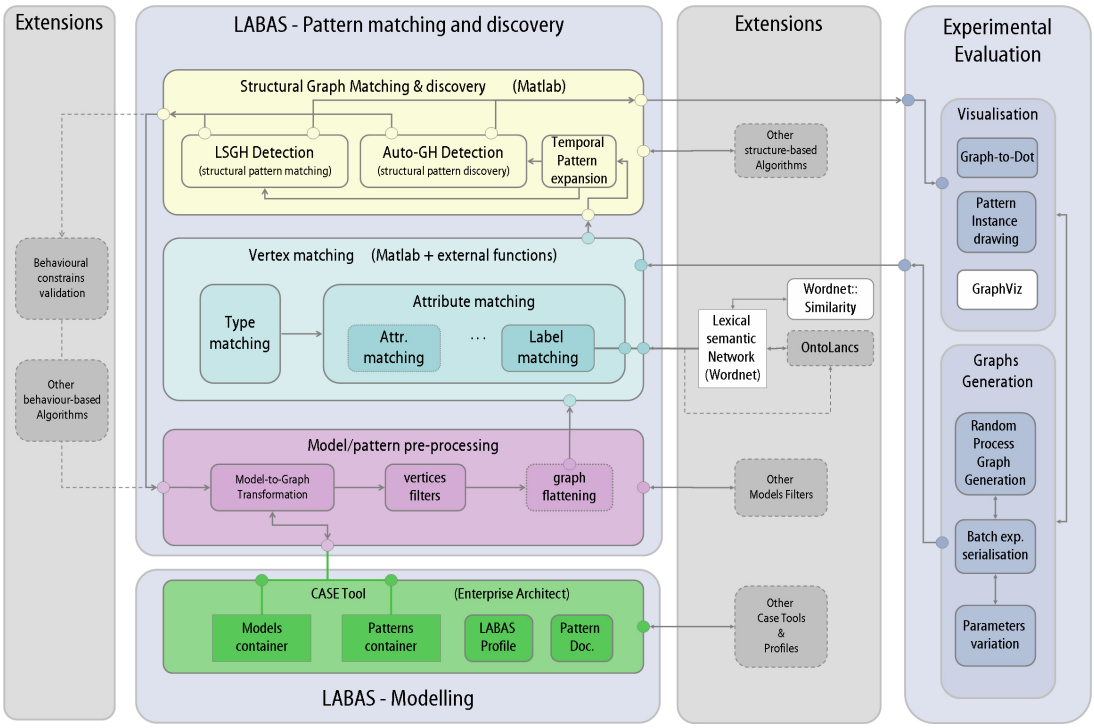


Figure 8.22: Tool chain support.

8.7.1 Matlab Functions for Matching, Discovering and Experimental Environment

The proposed techniques (algorithms) are implemented as a set of Matlab functions available as m-files[7]. Appendix D describes the main m-files pattern matching and discovery functions. A number of complementary functions to manipulate matrices and to provide the experimental environment are also described in the appendix. For example, they provide functionality to get the neighbours of a subgraph, to transform a graph from directed to undirected or vice versa or to visualise the results of a matching or discovery process.

8.7.2 Label Similarity

Semantic matching of label vertices is used during vertex matching. This functionality is external to Matlab. Initial experiments used WordNet::Similarity [Pedersen 2008] to obtain information about words (concepts) contained in a lexical semantic network (Wordnet [Miller 1995]) and for calculating similarity between words. Currently an extension using OntoLancs [Gacitua 2008] is being developed and its use is discussed in Section 10.3.2. OntoLancs allows to compose different NLP techniques in a single framework, which is suitable for several approaches that can be required during vertex matching in the LABAS framework. OntoLancs uses graphs described in GraphML [GraphML 2010], which allows a simple XML representation that includes attributes and types for graph vertices. The next section describes a function to generate random process graphs. The function, `matrand2`, was written in a Matlab script. For graphs generated with `matrand2`, a basic function to transform graphs created with `matrand2` to graphs described using the GraphML format was created for initial experimentations. The transformation function, `GraMat_to_GraML`, is included in Appendix D.

8.7.3 Graphs Generation and Visualisation

Part of the experimental evaluation uses randomly generated graphs. These are a special type of graphs whose vertices have a bounded in/out-degree and they have source and sink vertices representing initial and end process events. These type of graphs are so-called *random process graphs* in this work. A Matlab function (`matrand2`) generates random process graphs based on created adjacency matrices and type and attribute vectors representing process graphs. The function implements different ways to create edges between vertices. The default alternative creates an edge between each pair of vertices with probability $1 - (1/sz)$, where sz is the number of

[7] `www.computing.dcu.ie/~vgacitua/Thesis/mFiles.zip`

graph vertices. The function does not ensure dead-lock free graphs and in some rare cases it mights generate disconnected graphs. Graphs involving disconnected subgraphs are discarded. Cycles that can be potentialy dead-locks represent behavioral errors of actual process descriptions. It is relevant for a graph-based matching algorithm to be able to detect patterns on graphs that may contain this kind of error (non-deadlock free cycles) – likely to occur in reality [Awad 2008c].

An external graph visualisation tool can be used to graphically explore the randomly generated process graphs, pattern graphs and matched or discovered patterns in target graphs. In order to visualise the matched and discovered patterns in graphs maintained in the Matlab environment, the `matches_to_dot`, `testDiscovery1` and `graph_to_dot5` are used. `matches_to_dot` and `testDiscovery1` use `graph_to_dot5` to transform adjacency matrices and attribute/type vectors to a readable format for GraphViz (.DOT files). `graph_to_dot5` is adapted from the existing Matlab script (`graph_to_dot`) and it adds facilities to highlight matched edges and vertices in graphs. Details of the functions `graph_to_dot5`, `matches_to_dot` and references are in Appendix D. Figure 8.15 shows an example of matched patterns in a model graph visualised in GraphViz.

8.8 Summary

This chapter showed a set of experiments used to evaluate the effectiveness and efficiency of the proposed techniques for pattern matching and discovery, and a number of cases to demonstrate the support for end users and semantic aspects that strengthens the usability and feasibility of the techniques.

Experimental Evaluation. In Section 8.3, the accuracy of the structural aspect of the pattern matching technique (basis the discovery pattern technique) is evaluated. Pure accuracy based on the matching results and *match accuracy* [Melnik 2002] based on the number of required adjustments in the algorithm results are investigated. The results show that when trying to identify exact and complete instances of patterns in graph models (where partial matches are not allowed and only a one-to-one relationship between pattern roles and their instances are allowed) the algorithm's *match accuracy* for positive and negative samples converges to the ideal value (one) with the increase in size of the target graph, but (pure) accuracy decreases in value. However, since the algorithm is designed to identify partial matches and to allow a one-to-many relationship between pattern roles and their instances, in such conditions, the algorithm's *accuracy* and *match accuracy* meet the ideal value (equal to one). Nevertheless, it is important to note that both measures can be downgraded by the vertex matching step, where semantic matching can become relevant.

Section 8.4 described an experiment to derive an estimated measure of the pattern matching algorithm's processing time. The results showed that the algorithm's required time to solve the matching problem increases quadratically with the number of vertices in the target (undirected) graph (i.e., vertices and edges in a directed graph). The structure of matched patterns also influences the algorithm's performance. The results suggest the algorithm has advantageous performance characteristics for small and medium size graphs – up to 2000 vertices – that could be considered for run-time requirements, and it can handle patterns with cycle-like (or closewalk-like) structures, which are often difficult to detect within a reasonable processing time otherwise (see details in Sections 2.4, 2.5, 2.6 in the literature review chapter).

In practice, the results from Section 8.3 indicate that the proposed techniques can be accurate and simplify the pattern identification tasks that analysts and architects would otherwise perform in a hand-crafted fashion. Thus, the implementation of the techniques in tools can assist end users with automated support when performing pattern identification tasks during process model analysis and pattern-based services design. Moreover, considering the results from Section 8.4, it is expected that the techniques present advantageous performance characteristics for models whose size fall in the category of sizes for common process-centric models in organisations. For simple process element descriptions, the processing time could be considered for run-time requirements. However, complex process element descriptions would require additional efforts to create efficient semantically enhanced vertex matching techniques. Additionally, the algorithm required modest time to process pattern structures that are often considered rather time demanding such as cycle-like structures. This characteristics make the proposed techniques sufficient for design-time scenarios and promising for run-time scenarios.

Case for Semantic Vertex Matching. Section 8.5 presents a case to illustrate how the approach can handle the complexity of the vertex matching step when semantics is considered. An example involving the matching of labels described in natural language was presented. In general, to determine if a vertex in a model graph semantically matches a vertex in a pattern graph, the types of the vertices should be semantically matched and the similarities between common attributes of vertices should be aggregated. The case in Section 8.5 demonstrates how additional components adding semantic support to match vertices can strengthen the feasibility of the pattern matching and discovery techniques.

Tool Support. Section 8.7 provides an overview of the tool support provided for the matching and discovery techniques in the context of the LABAS framework.

Models and patterns created with the LABAS profile and pattern documentation facilities can be transformed to model graphs and pattern graphs used by the algorithms (see Section 7.3.1). Different modelling languages can be used adding additional model-to-graph transformation components. Each of these components takes a model described in a specific modelling language and transforms it into the graph representation used by the proposed algorithms. Also, additional structural pattern matching and discovery algorithms can be added to the tool chain support, as well as external support for semantically enhanced vertex matching and new components for pre/post-processing models. There are more opportunities to add additional elements to the proposed modelling and techniques environment. This provides a fertile base to continue with not only theoretical research but experimental research for graph-based architecture descriptions.

Chapter 9

Interviews: State-of-the-Practice in Process Analysis

Contents

9.1 Overview

A complementary assessment looks at aspects of the state-of-the-practice with regard to process analysis activities. Process analysis activities are critical to design solutions for enterprise process and application integration. The assessment consists of a set of interviews with practitioners whose roles are software architects, business analysts or managers in the IT domain. From results of the interviews, it is expected

to obtain a practitioner's opinion regarding the relative importance of process patterns and pattern identification techniques and tools to support them in the context of process analysis and systems design.

The interviews were conducted with IT professionals and a total of 19 valid interview forms were completed. While the interviewees are from a single country, their background and organisations are diverse (partly with strong international links); hence, the geographic bias that may affect the generality of results is diminished. Bias originated to a specific geographic scenario. Valid forms encompass information of 34 different organisations where the interviewees are or were working for. Organisations cover diverse industry domains, involving mining, health care, information technologies, government, professional services, insurance and retail.

The interview form, developed with the support of professional experienced in business and software process modelling, can be reviewed in Appendix E. The interview process involved a 15 minute presentation used to explain the purpose of the interview and to clarify the semantic of concepts mentioned in the interview form. The conduction of interviews was completed at the end of March, 2010. Quantitative results are associated to answers to a number of closed questions in the interview form. Note that results are biased to the interviewees' opinions and therefore, given the number of interviews, the assessment provides an *opinion* type of result. Open questions provide qualitative information and suggestions to the overall context where this work is positioned.

The next section explains the results of the interview process separated in results of closed and open questions.

9.2 Results of Closed Questions

The following figures and associated explanations refer to the results of closed questions. Some closed questions contain space to comments used to clarify the selected answers.

9.2.1 Profile of Interviewees and Organisations

Interviewee's profile. Figure 9.1.B shows a summarised profile of the interviewees containing their roles within organisations (question 1) and the results of an auto-assessment regarding their expertise for modelling *processes* and *process constraints* (questions 4 and 5). Interviewees are mostly managers and business analysts with medium- to advanced process-modelling knowledge, including process patterns. Note that it was agreed during the interview process that the term *process constraints* was an umbrella term capturing the ideas behind the concept of process pattern,

business process regulation, business process rule, best practices and procedure. The term refers to constraints on how processes are structured, how process elements are described (semantic) and constraints over data values associated to attributes from process elements. This umbrella term was used to facilitate communication and it captures the central concept of (process) patterns in this thesis.

Organisation's profile. Figure 9.1.A shows the distribution of the organisations' size and geographical locations (questions 2 and 3). The organisations are mostly large (more than 5000 employees) and have global presence. The organisational units where the interviewees belong to are directly (60%) or occasionally (40%) involved with process model analysis activities. Common process model analysis activities are explained to the interviewees prior to completion of the interview form.

Relation of the interviewee with process analysis activities. Figure 9.2 complements the previous Figure 9.1 with information regarding the interviewees' familiarity to (question 7) and involvement with (question 19) specific process analysis activities. Beyond the activities mentioned in the interview form, the interviewees referred to other activities during the analysis of process descriptions, among the most mentioned are process simulation and tests as feedback to process re-design. Process patterns were often related to best practices, to means for documenting procedures and to common rules and regulations across processes.

Relative importance of process analysis and design activities in organisations. Figure 9.3 indicates if the organisations consider or do not consider dedicated roles or activities to analyse and design processes and process constraints.

9.2.2 Process and Process Constraints Documentation and Notation

Sources of process model documentation. Figure 9.4 shows activities and initiatives that are sources of process documentation in the organisations where the interviewees are working or previously worked (question 8). Besides the options in the interview form, some interviewees commented that the design of new business processes, process externalisation and certification are other sources of process documentation. Given the nature of the documentation sources, it is expected that process model documents are at a high level of abstraction (further from implementation).

Textual versus graphical documentation. Figure 9.5 shows to the percentages of textual versus graphical processes and process constraints documentation. A Likert scale type from one to five is utilised – where one indicates only textual and five

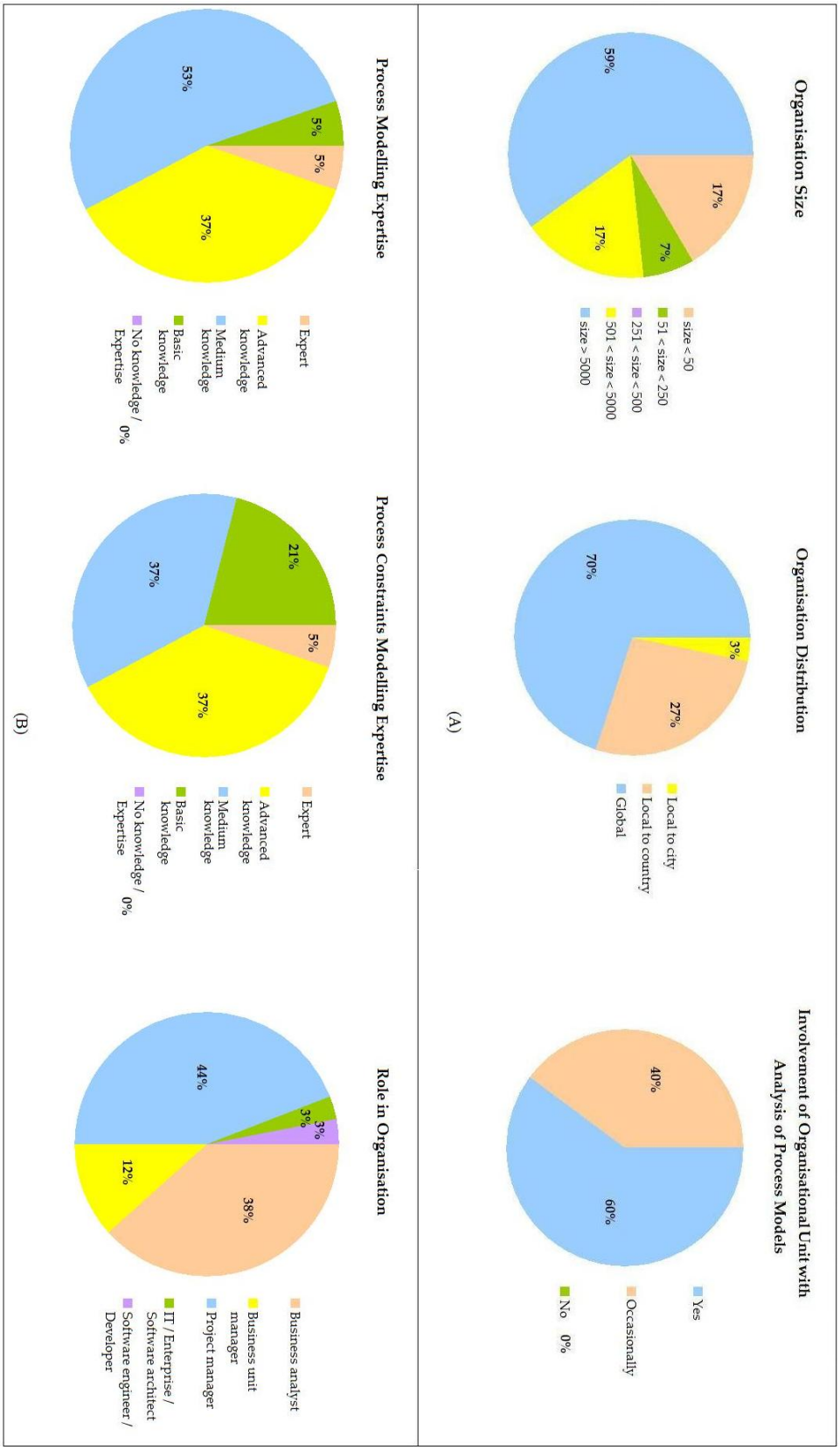


Figure 9.1: Organisation and interviewees profiles.

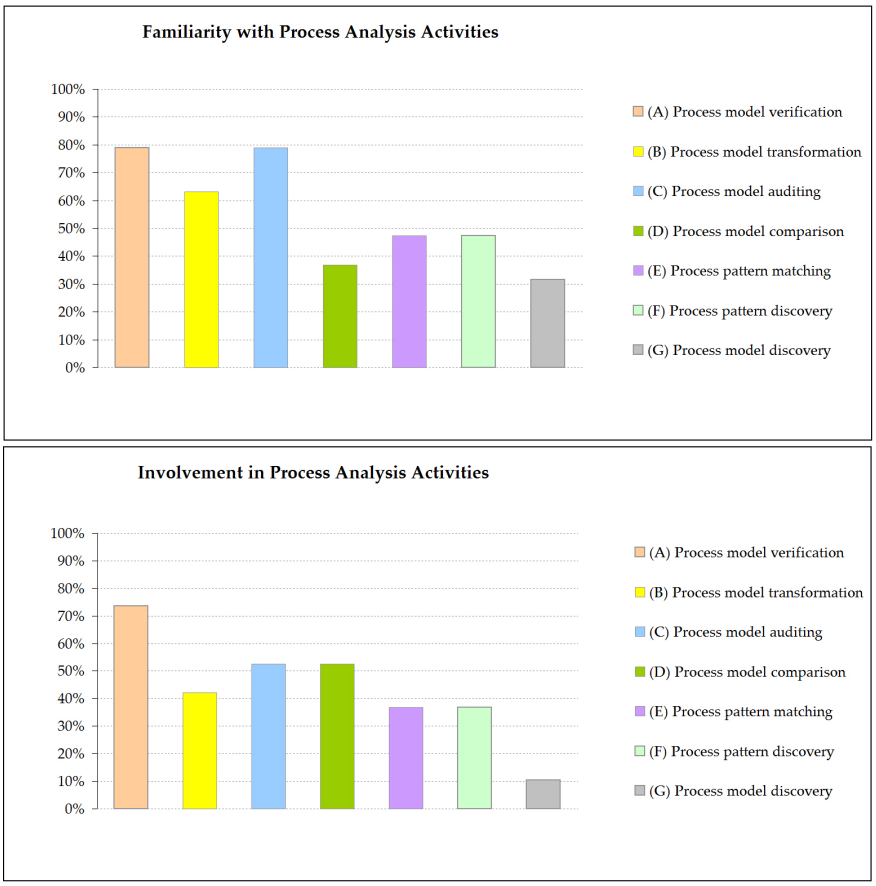


Figure 9.2: Interviewees' familiarity to and involvement with process analysis activities.

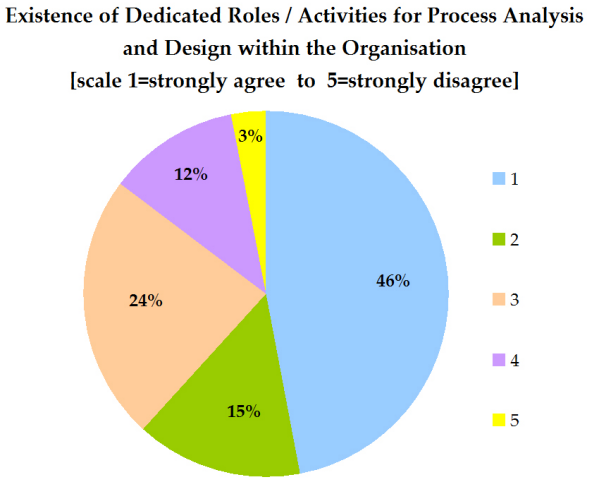


Figure 9.3: Existence of dedicated roles/activites for process analysis and design.

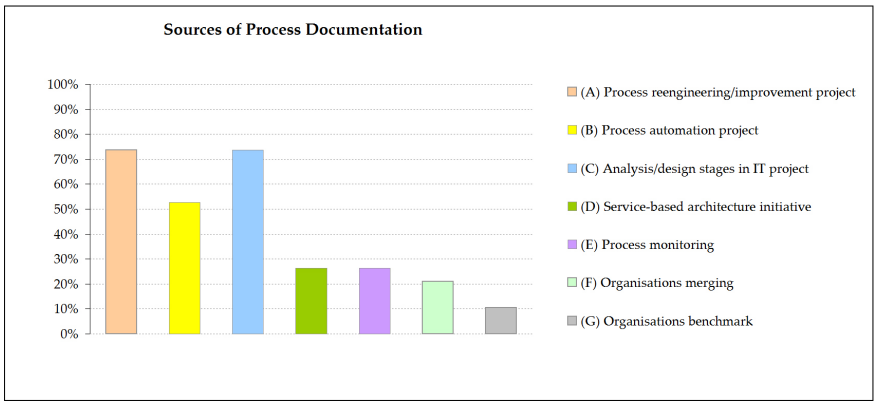


Figure 9.4: Sources of process documentation.

indicates only graphical. The results show that often there is a mixture between textual and graphical documentation, where textual documentation predominates over graphical one. This results suggest that the techniques proposed in this work should reinforce the semantic analysis aspect to support analysis of textual documentation.

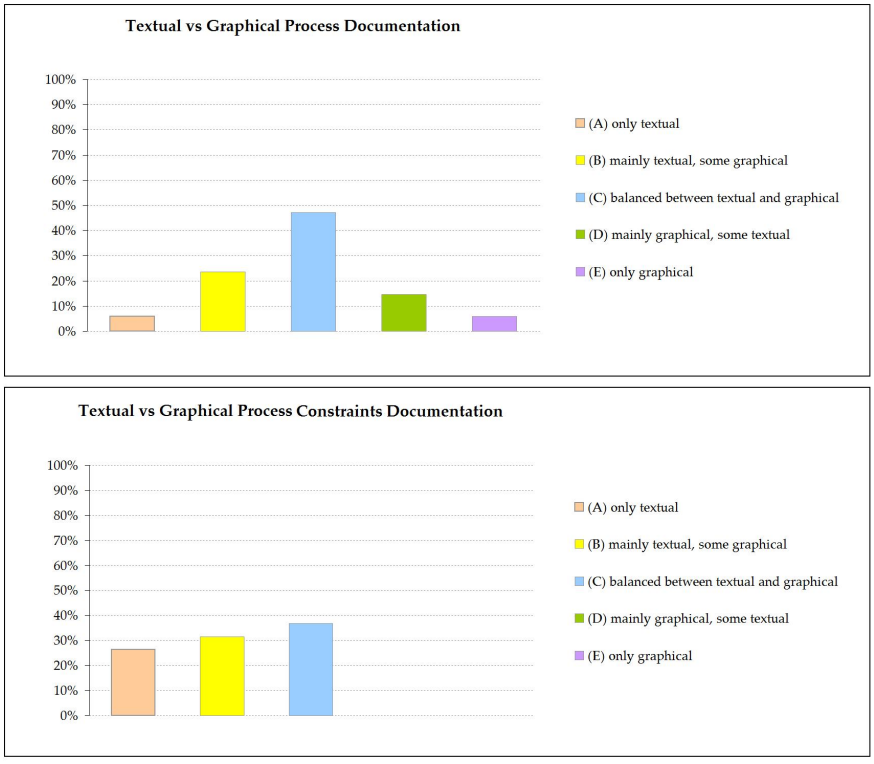


Figure 9.5: Textual versus graphical documentation of processes and process constraints.

Graphical modelling notations. Graphical documentation can involve the use of different process modelling notations. Figure 9.6 indicates the percentages of utilised modelling notations (questions 10 and 17). Among the options are BPMN, UML activity diagrams, EPC, IDEF0/3, graphical support for WS-BPEL. Enhancements to these notations to capture process constraints were provided as part of the options for process constraints documentation. Beside the options provided in the interview form, the interviewees indicated that they also use more informal graphical notations based on boxes (representing process elements) and arrows to connect them. For all indicated notations, there are ways to transform them to graph-based process documentation.

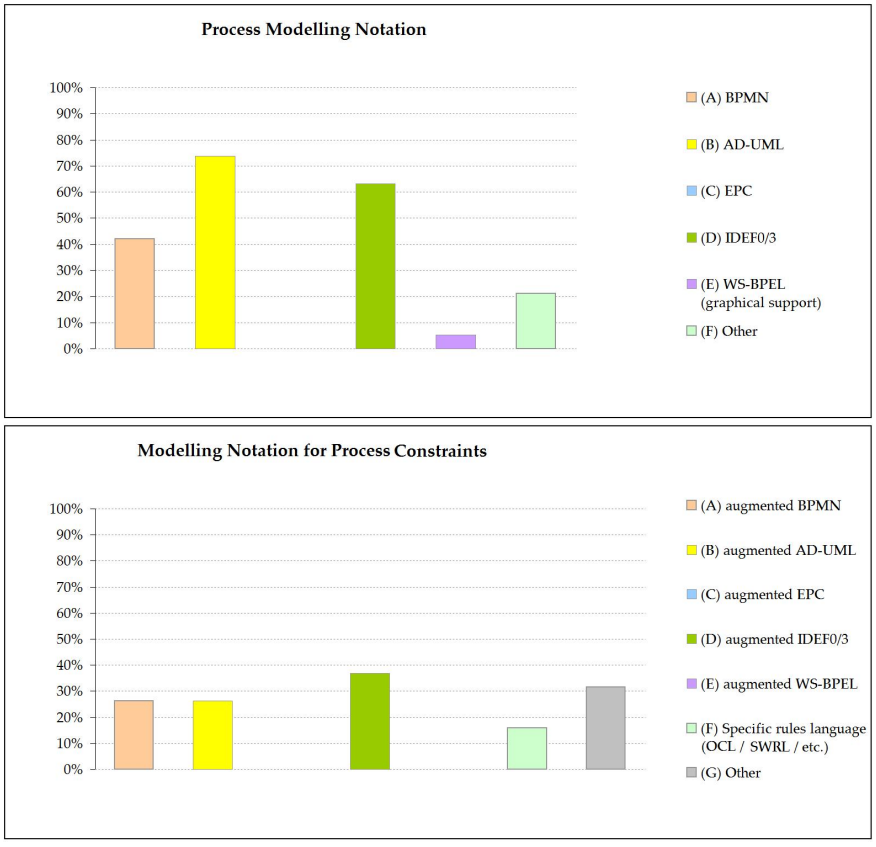


Figure 9.6: Modelling notation for processes and process constraints documentation.

Complexity of graphical process models in terms of its size and number. Figure 9.7 refers to estimations in the amount of activities per process model document and the amount of process model documents in the organisations where the interviewees are working or previously worked. More than 75% of the organisations have between 5 and 50 process model documents, while only 15% have more than 100 process model documents. Almost 60% of the organisations have process model documents

containing between 5 and 15 activities, while almost 40% have more than 15 activities per process model document. The results indicate that, in general, the complexity of process models in organisations is relatively moderate with commonly 5 to 20 process model documents with 5 to 10 activities per document. For this work, target organisations for the proposed techniques would be those in the upper results – i.e., organisations with (approximately) more than 100 process model documents and more than 15 activities per document – involving approximately 1500 elements (at the minimum) to be analysed.

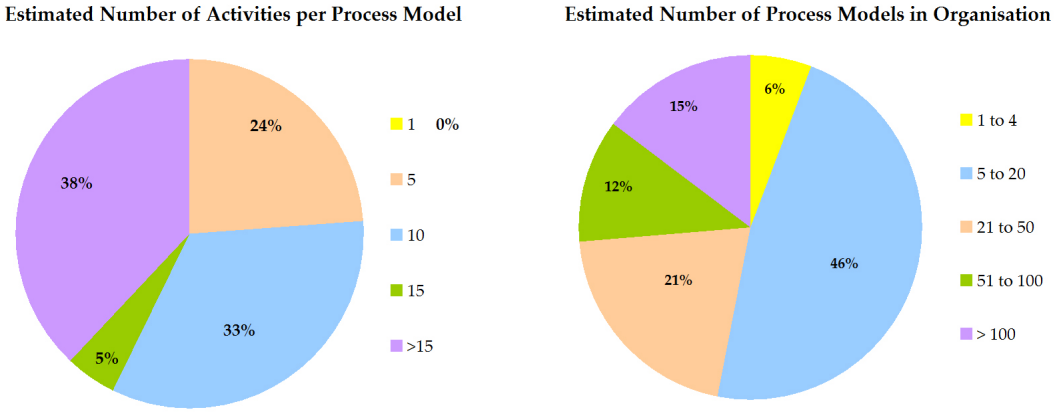


Figure 9.7: Estimated number of process model documents and activities per document.

9.2.3 Compliance with and Type of Process Constraints - Including Process Patterns)

Obligation to comply with process constraints. Figure 9.8 refers to the estimated frequency (from a daily basis to never) to which employees are required to comply with process constraints.

Types of process constraints. Figure 9.9 shows the percentage of process constraints types considered by the interviewees. Four types of constraints were indicated in the interview form. Structural constraints, involving for example the ordering of process elements; constraints in data values associated to attributes in process elements, for instance, the cost of an activity cannot be more than a specified amount; time constraints, for example, an activity or decision can not be longer than a specified time period; and semantic constraints, restricting the meaning of process element types and attributes according to specific domains. The latter can be associated to the type- and attribute- sets limited to and structured according to a determined ontology.

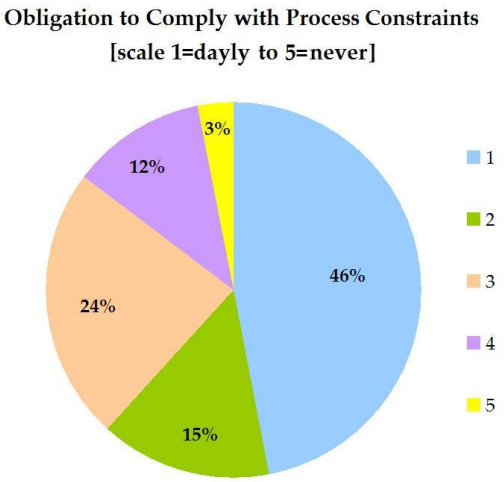


Figure 9.8: Obligation to comply with process constraints.

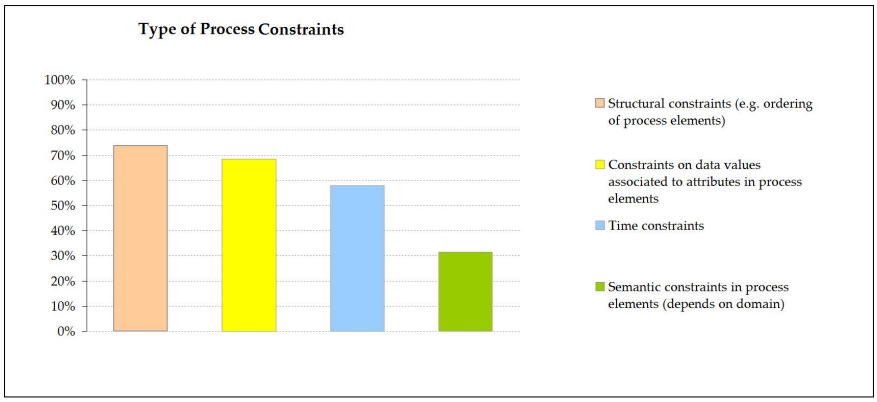


Figure 9.9: Type of process constraints.

9.2.4 State and Relevance of Automated Process Analysis

Automated process analysis activities. Figure 9.10 shows the degree of automation of analysis activities performed by interviewees. A Likert scale type from one to five indicates completely automated to completely manual, respectively. While most of the activities are manually performed, process model verification and auditing are in a few cases completely automated. Two answers indicate usage of pattern matching and discovery with a high degree of automated support. One of answers refers to specific sets of process constraints related to security risks that can be interpreted as domain-specific process patterns. For the other answer, the interviewee did not provide comments on how automation was achieved. Also, process simulation was one of the additional activities seen by the interviewees as an activity possible to automate, providing scores between three and five.

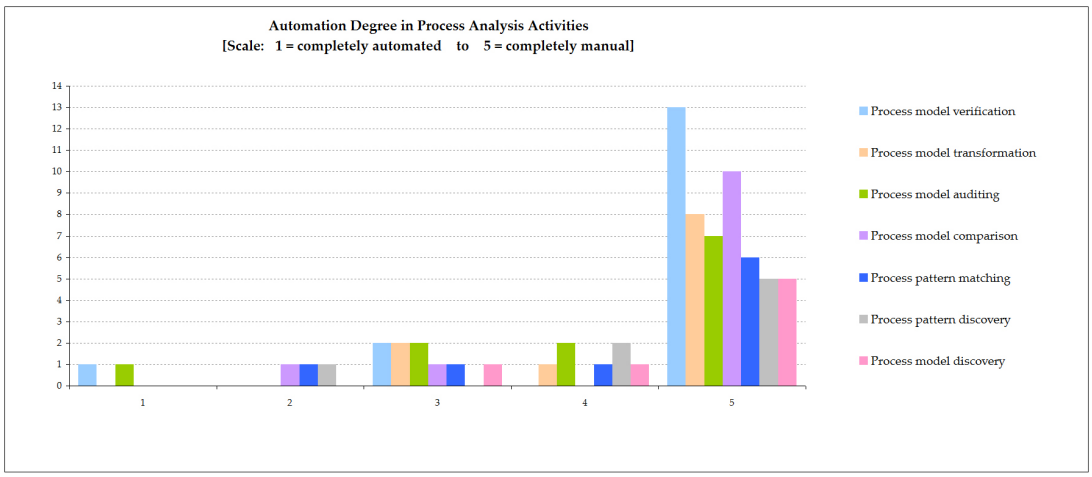


Figure 9.10: Degree of automation in process analysis activities.

Finally, Figure 9.11 shows the interviewees' opinion regarding the possibility to automate (to some degree) the previously mentioned process analysis tasks. While most interviewees strongly agree, some of them indicated that in the organisations where they work or previously worked, there is not enough maturity and expertise regarding concepts related to process modelling, and therefore they think tools would be useful only after people in the organisation have the expertise to use them.

9.3 Results of Open Questions

The next sections refer to answers for questions 14 and 21 to 24 in the interview form. They involve comments regarding the factors influencing non-compliance with process constraints – including gaps between process patterns and models (question

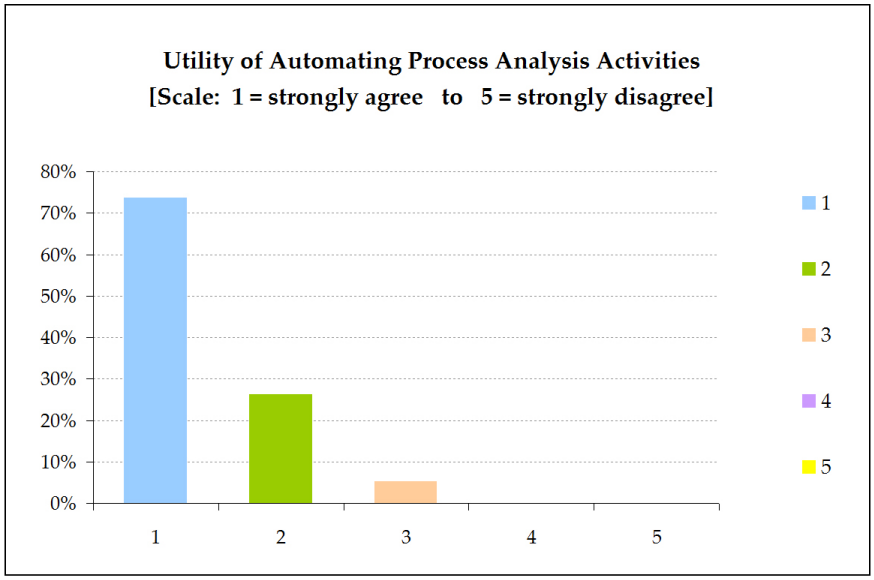


Figure 9.11: Utility of automating process analysis activities.

14), comments on general benefits of automating process analysis activities (question 21), specific comments on benefits of automating pattern matching and discovery (question 22), comments on other relevant activities that can be automated in the context of process analysis (question 23) and open and general opinions regarding process analysis (question 24).

9.3.1 Factors Influencing Non-compliance with Process Constraints (Question 14)

Most interviewees referred to business rules and best practices at the process level at the time of using the umbrella term *process constraints*. The umbrella term includes the concept of process pattern as defined in this work. Best practices can be documented as process patterns (see for example the case study in Section 8.5). Business rules can refer to constraints in the process structure, the semantic of process elements and attribute values [Lu 2008], [Awad 2009]. For instance, constraints to attribute values in process elements such as decision points – controlling the flow of information in a process – are commonly related to the business rules term. Business rules applying to business processes in the context of process compliance can be associated to process patterns and anti-patterns [Kharbili 2008] (e.g., the compliance patterns in [Ghose 2008], and the patterns/anti patterns for compliance in [Awad 2010]). Hence, the comments in this section apply to the process pattern notion of this work.

Almost half of the interview think that process constraints are often defined in-

formally and not enforced or embedded in existing processes, even those already automated (e.g., in workflows). Also, even if process constraints are in some way documented or more formally enforced, changes affecting them are not always implemented opportunely and, therefore, the original enforced constraints tend to lose their utility. A representative answer for this generalised opinion is one from a project manager from the IT industry:

> *Almost all process constraints are implicit in the organisation culture or defined as informal best practices among employees; they are commonly overlooked or satisfied on an ad-hoc basis. Few process constraints are defined formally and they sometimes introduce delays in the process due to increased workload. In some cases the cost of non-compliance is comparable or less than the cost of delaying the process.*

Note that some other answers indicated that there are justifications for maintaining process constraint documentation informal. For example, the opinion from a manager in the mining industry:

> *Some process constraints may have exceptions to be satisfied, which are defined in an ad-hoc manner by high-level roles in the organisation. This is normally not documented or embedded in automated processes. They (exceptions) often have a strategic origin.*

Other two frequently mentioned factors for non-compliance (more than a third of the interviewees mentioned them) are:

- There is a believe among employees that there is a high correlation between increase in workload and increase of compliance level with best practices and business rules. Non-compliance is justified by reductions in process delays and improvements of the process performance.
- Existing process constraints were defined without a deep understanding of the actual processes and, therefore, they can be often more restrictive than what can possibly be implemented in practice. This is common to process constraints defined by organisational roles or units that do not perform the process. It is relevant to identify the root causes for non-compliance.

Other mentioned factors indicate that:

- For automated processes, a non-adequate integration between the process execution engine and the business rule engine can influence the satisfaction of implemented business rules negatively.
- Time constraints are often violated in some health care institutions due to execution of redundant tasks.

- In some organisations, constraints are associated to roles instead to processes. Compliance with constraints would depend on incentives and sanctions defined for organisational roles in relation to the satisfaction and violation of constraints.
- Process constraints that are not enforced in processes and rely on the employees responsibility to comply with them, normally create increased costs of training in organisation with high levels of staff rotation and low levels of compliance.
- Frequent changes in business rules and non-automatic updates make them more difficult to enforce in implemented processes.

9.3.2 Comments on general benefits of automating process analysis activities (Question 21)

A generalised opinion is that tool support helping to automate process analysis activities can reduce time and human errors. Emphasis on model verification and auditing was frequent among the answers. Also, more than half of the interviewees indicated that there is a lack of process documentation in organisations and therefore tool support for analysis activities would be more appreciated when large volumes of documentation are available. Specific comments on pattern matching and discovery activities were discussed in more details in the next question.

9.3.3 Specific comments on benefits of automating pattern matching and discovery (Question 22)

This question specifically target the opinion of the interviewees with regard to the benefits of automating pattern matching and discovery tasks. The benefits frequently mentioned among interviewees, ordered from higher to lower frequency, are:

- Automating pattern matching can help to identify deviations from norms and procedures defined as process patterns. This was specially emphasised for regulations in health care institutions (AUGE) and several organisations involved in certification processes that involve compliance to standard processes (e.g., ISO, ITIL).
- Automating pattern matching and discovery can help to identify opportunities of reuse at high- and low-level processes. This can reduce costs of supporting software by avoiding redundant implementations. The domain of reuse ranges between different abstraction levels. It was mentioned that there are common business processes across organisational units, also similar low level implemented processes (e.g., similar composite services across software development projects) and similar processes trying to implement a standard process

(best practice) across companies in the same industry domain. Among the domains mentioned by interviewees were mining, health care, retail and insurance domains.

- Automated pattern matching and discovery was defined as a critical element to facilitate process improvement through re-design. In several occasions, it was mentioned that tool support to simulate or analyse the impact of changes on a pattern-based re-designed process would be of great utility. Process simulation would provide early information about possible process re-designs. A re-designed process would incorporate patterns that previously were only partially instantiated and possibly would involve the elimination of redundant process sections associated with discovered patterns. Feedback between a pattern matching and discovery stage and a simulation stage would converge to an improved process design.

  One interviewee mentioned a process re-design initiative based on discovered patterns – in that case named frequent sub-processes – that took a significant amount of human resources and time. Tool support after process documentation would have been clearly appreciated.

  Another interviewee mentioned that reuse of automated low-level processes (in this case implemented as web services) across software development projects was important in the selection of strategic partners for software development. The organisation works with external companies to develop the software support for the business operation. External software development companies could receive a higher payment if they do not reuse previously implemented services and charge for new implementations. Companies willing to reuse previously implemented (process-centric) services by arguing they could improve delivery times were considered of a higher strategic value. So, process-centric service reuse was seen as an strategic value for the partnership and the long-term relationship between provider and client companies.
- The interviewees referred to the identification of *bad* practices (or *anti-patterns*) an activity as relevant as the identification of *best* practices. However, there was no specific agreement on how these bad practices (or anti-patterns) could be discovered.
- Automated pattern matching and discovery should be accompanied by tool support for patterns storage and organisation.
- Pattern discovery could help to identify internal best practices, but also redundancies.
- The more complex and large the processes being analysed are, the more useful the tool support for automated pattern matching and discovery is.
- It would be more useful to discover recurrent process events associated to im-

proved business objectives (e.g., sales, reduced costs) than discovering redundant process structures.
- It would be helpful to obtain automated suggestions for where else a recently implemented pattern could be applied.
- It could be possible misleading if the documented processes were not correct or far removed from reality.

9.3.4 Comments on other relevant activities that can be automated in the context of process analysis (Question 23)

Comments indicate other relevant activities that can be automated in the context of process analysis but that were not mentioned in the interview form (see mentioned activities in Figure 9.2). Comments, ordered from higher to lower frequency, are indicated below.

- Process simulation facilities to investigate changes in processes before implementation. Also, feedback from possibly monitored processes.
- Monitoring of key variables associated to processes and the consolidation of these variables in aggregated reports that summarise the dynamic of process variables in determined periods of time. A special emphasis was viewed for reports associated to process costing and its utility to define budgets for software support and development. A desired characteristic for process monitoring facilities is the possibility of monitoring multi-dimensional goals.
- Automated evaluation of risk at process level by dynamically identifying events patterns.
- Automated process discovery by tracing back relevant steps to create products. Instrumentation would focus on steps relevant to the products creation more than all transactions associated to information flows in processes. This could reduce the amount of logs used to discover processes, but could loose information that can reveal process inefficiencies, such as frequent cycles.
- Support to increase automation during the refinement of high level processes to implementation (executable processes).
- Automated generation of interrelated process steps influencing the values of specified process metrics.
- Support to increase automation to transform process constraints specifications to implemented process constraints in automated processes (e.g., in workflows). A standard language closer to natural language is needed to express process constraints if the organisation wants to make them widely understandable among employees.

9.3.5 Open and general comments regarding process analysis (Question 24)

This last question allows interviewees openly express ideas regarding their general view on process analysis, even if they were not related to automation or tool support for process-centric software design. Comments are in no particular order:

- A detailed methodology for process modelling is missing. Recommendations that define, for example, until what level of abstraction it is necessary to document a process in order to be helpful to the final goals (e.g., process implementation as a workflow, process re-design to reduce costs).
- Documentation of processes is scarce in many organisations. Also, the quality of documented process models is not always sufficient. Often a process model reflects a desired process more than the actual process and hence, the identification of weaknesses and strengths in the process can be erroneously derived and interpreted.
- Process analysis tools should be available at low cost and designed not only for business analysts and architects but for the actual performers of processes. A widespread use of these type of tools would increase the amount of process documentation and, even more important, the provision of automated support for process analysis.
- In some organisations, the cost of generating process documentation is unfortunately too high to be justified in comparison to other needs. This is the situation for several health care institutions.
- Documentation for a single process but generated by different participants requires tools that help with their consolidation into a single process model.
- Several software development projects have a more expensive domain analysis stage than a developing stage. Automated support to reduce time and costs in the domain analysis stage is critical for this type of project. This includes process analysis.
- Automation of any type of process analysis activity would be relevant in organisations that are more process-oriented. A number of organisations are more role-oriented and the analysis occurs at functional level. Less emphasis is given to flows of information or processes.
- Automated upgrades of process models after changes are basic to maintain the value of process documentation. A few non-upgraded models render the entire set of process models invalid.
- It would be useful to integrate two different kinds of process modelling tools, one covering information flow-based processes and other targeting physical process modelling (e.g., modelling of fluids, chemical reactions, material trans-

portation). This is important in industries that have interrelated productive and administrative processes, such as in the mining industry.
- Identified patterns (or best practices) require an effective medium to be shared and reused. Similar to the ideas of service directories, process-level patterns should have a (central or distributed) location were they can be published and reused.
- The execution of automated processes separate from business rules enforcement can be relevant to improve the impact of changes in business rules over the maintenance of implemented processes.

9.4 Summary

This chapter provides the results of a complementary assessment consisting of conducted interviews to a number of analysts, IT architects and managers in the IT domain.

Interviewees and organisations profile. This work mainly targets large and distributed organisations that can more likely have challenges regarding processes and applications integration and possible considering service-centric integration solutions. Most of the interviewees are working or have worked for large and distributed organisations, where the software support is heterogeneous and where needs for process automation and compliance to regulations exists.

In general, the interviewees were familiar or directly involved with process analysis activities. Although their expertise in process matching and discovering tasks were less than other activities such as process model verification, they recognised the benefits of performing (and automating) these tasks.

Process and process constraints models. The existence of graphical process model documentation was not always predominant in contrast to textual support, making more difficult any attempt to automate process model analysis tasks. However, the interviewees referred to the increased interest of documenting processes graphically, for instance, to facilitate the comparison to standards or recommended (process) practices, and the opportunities that process discovery tools can provide to trace back relevant process steps to create products – which also generates graphical, often large, process model documentation.

The graphical notation used to model processes are all graph-based notations and it is possible to translate them to the graph-based representations referred in previous chapters. This supports the feasibility of the proposed approach in real scenarios. The complexity and size of process models can be considered medium,

with most organisations dealing with at most 750 activities in process models (50 models with 15 activities). However, they recognised that the process models often were related to high abstraction levels and implementation levels involved many more process elements that were not documented. This supported the opinion that process documentation should be improved.

Summary of comments to open questions. From the negative and positive opinions related to the open questions, several aspects can be distinguished.

Negative comments in question 14 have possible solutions. This provides an opportunity to deliver findings and proposals in academia to the industrial environment. For example, a more rigorous and formal documentation of process constraints (for instance, in the form of process patterns and anti-patterns) can help to define more unambiguously process constraints. Its implementation and enforcement in processes can go from high-level designs used in the strategic definition of the business operation to lower implementation levels. At this level, mentioned problems such as a deficient integration between the process execution engine and business rule engine is a purely technical problem that can be addressed at application integration level. Maintaining processes and their constraints enforced can be facilitated with process automation and run-time detection of process constraints violation. Automated process (anti-)pattern identification can assist the detection of process constraints violation and also foreground opportunities to apply a desired pattern. The future work section (Section 10.3.2) refers to extensions of the proposed pattern identification techniques to a run-time scenario. Here the focus was on design-time concerns. Additionally, updating process constraints automatically can be supported by an adequate environment for process constraints documentation, with linkages and mediums to propagate changes across involved process elements and its software support.

A generalised opinion in question 21 is that tool support for automating process analysis activities can reduce time and human errors. This support the hypothesis and proposed approach of automating some of the process analysis activities (process pattern matching and discovery) to benefit maintainability by reducing human errors and time during process analysis.

Specific comments with regard to the benefits of automating pattern matching and discovery tasks (question 22) strongly support the hypothesis of this work, specially when related to compliance and reusability. For example, the interviewees mentioned that by automating these tasks they could be assisted during the identification of process deviations from norms and procedures, they could obtain suggestions for where else a recently implemented pattern could be applied, and if there is redundant software support associated to similar processes helping to reduce costs.

The answers from question 23 provide comments on other relevant activities that can be automated in the context of process analysis were provided. An interesting line of future research based on these comments would be pattern-based process simulation and monitoring in a close-loop involving process analysis with simulations, process modification, implementation, monitoring and analysis again. This could be considered in a design-time setting but also in a run-time scenario, where process adaptation towards a desired process state could be assisted. More details are provided in the future work section of the next chapter.

Thanks. The information provided in this chapter was possible thanks to - in alphabetical order - Paula Aceituno, Alicia Balbontin, Miguel Barrera, Jorge Carvallo, Fabiola Castillo, Maricel Contreras, Felipe Cors, Cristian Diaz, Sandra Estay, Medardo Folch, Carolina Martinez, Pamela Montanares, Patricio Moya, Ezequiel Munoz, Ramiro Pinedo, Claudia Reinoso, Viviana Silva, Romina Torres and Hernan Vidal.

Chapter 10

Conclusions

Contents

10.1 Overview

This chapter explains the main conclusions of this work and it is separated into two parts. First, a summary of the contribution is provided, including references to the aim and focus, the probe of the hypothesis and practical implementation. Second, a discussion with references to the assumptions taken into consideration, extensions and application of the achievements to other domains and descriptions of possible further research are provided.

10.2 Summary of the Contribution

10.2.1 Relevance and Focus

With the increasing need to integrate applications and enterprises as a response to globalisation building up on interoperability platforms, the focus has been recently on integrating BPM and SOA in an effort to provide a joint view on business activities and their underlying IT architecture.

The aim of this work is to contribute techniques to automate and support the reuse of process-centric architecture and model abstractions to improve the current

approaches for enterprise SOA design and integration. Guidelines and techniques supporting enterprise SOA design for processes and applications integration has a practical impact in organisations, especially benefitting those that have to deal with the complexity of large and heterogeneous enterprise-wide application architectures that support their internal and inter-organisational processes. Business operation (processes) improvement is the visible result of solving the integration problems and it is what organisations are interested in. Focusing on processes and linking them to software levels directly targets the organisations' concerns. Automation and abstraction were the principles followed to contribute to improved SOA design and integration methods and techniques. Abstraction at a process level was central in this work and it was realised in the form of process patterns. Automation of process pattern matching and discovery provides concrete support that can be implemented in tools assisting analysts and architects during the development of enterprise SOA design and integration solutions.

10.2.2 Achievements and Practical Implementation

Throughout the development of the proposed LABAS framework and pattern-based techniques, and the use of a scenario-based method with case studies and the conducted experimental evaluation, the initial hypothesis was demonstrated. It was shown that pattern-based techniques used in layered service process models can benefit maintainability, functional suitability/compliance and traceability of service-based systems for enterprise processes and application integration. It was demonstrated in an experimental evaluation that process pattern matching and discovery can be automated while being effective and efficient. Cases showing how the techniques handle semantic descriptions of process elements and the implemented tool support strengthened aspects of feasibility and usability. Finally, interviews conducted with practitioners from diverse industry domains reaffirmed the relevance of the overall focus on processes and automated support during process analysis and process-centric design.

On the other hand, the pattern matching and discovery techniques, which in this work were implemented as a prototype chain tool, are possible to implement as part of open source or commercial tools. A concrete example is the development of a plug-in for Enterprise Architect tool [SparxSystems 2010]. The tool can be used with the LABAS framework and patterns can be documented and maintained in a repository managed by the tool. The plug-in would extend the tool's functionalities with support to match patterns from the repository in working models and to discover new ones from these or other stored models.

10.2.3 Reference to Background Research and Related Work

The background research and related work in Chapter 2 described approaches and findings from where this thesis builds up on. Existing SOA design approaches such as SOMA [Arsanjani 2004] and MSOAM [Erl 2004] can be considered as guidelines that structured the proposed LABAS framework. They can benefit from modelling support and explicit traceability provided in LABAS (similar to the work in [Zhang 2008]) and also from the solutions to document, identify and exploit process patterns as a medium to reuse design knowledge, complementing the use of solution patterns as in [Zhang 2009].

Since decades, literature related to process reengineering and management has considered the idea of reusable process designs associated to recurrent problems in the operation of businesses (see e.g. [Malone 2003]). Process patterns in this thesis capture such an idea in the form of reusable modelling (and architecture) elements at the process model level. Only recently, patterns at this level (processes) have acquired relevance in software engineering. This is emphasised in the context of process-centric SOA and BPM.

A number of recent contributions defining business process level abstractions such as *action* patterns, *workflow* patterns, *activity* patterns were described in Section 2.3.2.2. The concepts of *activity* patterns based on recurrent functions in [Thom 2009] and *actions* patterns in [Smirnov 2009] are close to the concept of *process pattern* in this work. Additionally, the work in [Smirnov 2009] provides a solution for uncovering *action* patterns in large data collections using association rule mining techniques and considering information in activity labels. Their aim is to facilitate the use of patterns to suggest additional actions for a process model.

Reuse of process patterns is a central aspect of this thesis. A solution to identify process patterns in concrete process models was used to aid the definition of new services. Graphs, as a natural representation for process models [Aguilar-Saven 2004], [Corradini 1996], were the basis to formalise the problem of identifying known and unknown process patterns in concrete process models. Attributed typed graphs abstract process models and patterns allowing a process pattern identification solution to be independent of concrete models and graph vertices to be typed and include several attributes (e.g., a label, cost, processing time). The problem of pattern identification (matching and discovery) is thus abstracted to a problem of subgraph matching and frequent subgraph discovery. Previous works on these two general problems on graphs provide a foundation to the proposed pattern matching and discovery algorithms. Section 2.6 describes a number of existing graph matching and frequent subgraph discovery techniques. The algorithms proposed in this thesis extend structural-oriented solutions such as in [Bunke 1997] and [Valiente 1997] with

semantic matching at vertex level. This is necessary to capture complex descriptions of process model elements. Additionally, this thesis considers inexact matching and pattern elements to be related to more than a single element in a process (beyond isomorphism).

10.2.4 Detailed Contribution

This work contributes techniques for pattern matching and discovery as a mean to provide automated support for the identification of process-centric abstractions involved in enterprise SOA design and integration. The focus has been on the analysis of process-centric models. A pattern-based and layered architectural framework was proposed to organise and maintain aligned modelling elements involved in the analysis and design stages of enterprise processes and applications integration projects based on service architectures. In detail, this work proposed:

- A layered architecture (LABAS) to organise modelling elements and patterns involved in the integration of enterprise processes and applications. Patterns can be documented using pattern templates containing graphical models of pattern configurations. Techniques for using patterns at different layers of the framework are introduced – pattern recommendation, comparison, modification, instantiation, combination, matching, discovery and pattern-based model refinement. Pattern matching and discovery are further developed.
- A UML profile to model processes, service architectures, application architectures and their interrelations in the context of the LABAS framework.
- A traceability model to maintain trace links between models elements and between model and pattern elements.
- A graph representation for process and process patterns used by the proposed algorithms for pattern matching and discovery.
- An algorithm for exact process pattern matching able to find complete and partial matches in process graphs.
- An algorithm for inexact process pattern matching able to find complete and partial matches in process graphs.
- An algorithm for hierarchical process pattern matching in process graphs.
- An algorithm for process pattern discovery in process graphs.

10.3 Discussion and Future Work

The previous section referred to the contribution of this work to assist the development of service systems for process and application integration systems with automated support for pattern matching and discovery. There are assumptions (which

can become possible limitations) and several directions to complement and extend the results in this work. This section describes some of them.

10.3.1 Discussion

Assumptions. Some of the assumptions that researchers can take when focusing on models, architectures and solutions to automate tasks related with model analysis are that models are available, they share a common way to represent their elements and they are possibly large and complex. Since this work aims to be applied in a business enterprise environment, the target models are those available in organisations. Although the experience collected from a number of practitioners in industry (Chapter 9) indicates that currently the availability of models in enterprises is rather limited (in terms of the amount of available models, their formality and complexity), opinions of the interviewed practitioners and observations from industrial reports such as [Palmer 2009] and [Oracle 2008] indicate that there are positive trends for an increasing availability of models and a more active use of these models as instruments to plan and lead the design and maintenance of enterprise architectures. Moreover, the increasing availability of tools for monitoring and storing information about the actual operation of enterprises - e.g., process mining tools [Aalst 2007], [Aalst 2009a], [Hill 2009] make process models (that can be large and complex) more and more available.

In addition to model availability, this work assumed the existence of architecture and model abstractions represented in the form of patterns. The focus was on process-centric patterns. Even though the use of process-level patterns has been notably emphasised in recent years, e.g. [Barros 2007], [Thom 2009], [Smirnov 2009], tools providing support to work with process-level patterns is limited [Gschwind 2008]. This work proposes a framework for service-centric processes and applications integration in which patterns receive a central role. Patterns can be documented, identified and explicitly traced to elements in concrete models where they are instantiated. However, more effort needs to be made to create the necessary pattern documentation. This work assumes the existence of documented patterns and the adoption of modelling facilities to create more pattern documentation in organisations. The complexity of pattern descriptions and their possible heterogeneity would influence the effectiveness of the proposed pattern matching and discovery techniques. In particular, heterogeneous descriptions of model and pattern elements and the complexity of their semantics would affect the accuracy of the matching algorithms and the associated processing time.

Extensions and Application to other Domains. The conducted interviews to practitioners from diverse industry domains confirmed the relevance of this work in a wide organisational spectrum. Opinions from practitioners regarding process analysis tasks beyond the objectives of this thesis - for instance, process monitoring and simulation - provide opportunities to extend the scope of this work to problems whose solutions also have a practical impact.

Other visualised extensions during the development of this work are explained in the future work section later on this chapter. They involve behaviour-based pattern instance verification, dynamic pattern matching and discovery, graph clustering for pattern discovery, pattern-based model augmentation and transformation, extended semantic similarity for vertex matching and the development of a process model generator.

The results obtained for the proposed (graph-based) pattern matching and discovery techniques can be transferred to other domains that use graph representations for models. Graphs are a more intuitive and powerful representation that have been used in numerous domains. Beyond the focus on processes and application integration problems, there are other technologies and paradigms such as cloud computing, programming by demonstration and ontology evolution where the use of graphs and techniques for pattern matching and discovery could provide beneficial results. Patterns in these domains can represent common and successful service compositions (cloud computing), knowledge from users recorded in computer-user interactions (programming by demonstration) and common ontology changes (ontology evolution). The suggested benefits are reuse and scalability through exploiting the principles of abstraction and automation. The main challenges would be addressing the dynamic nature of graphs in these domains and the semantic involved with graph elements descriptions.

10.3.2 Future Work

This section describes some directions of future work categorised as ideas for:

- *Improved semantic matching and application of patterns* - extension of pattern matching with enhanced semantic vertex matching, pattern-based model augmentation and transformation, behaviour-based pattern instances verification and the development of a process model generator to provide a publicly available corpus of process-centric models for comparative analysis;
- *Dynamic pattern matching and discovery* - adaptation and extension of approaches for community detection, diffusion in networks and classical graph clustering techniques to an scenario of dynamic processes; and

- *Process simulation and monitoring* - process monitoring with adequate instrumentation, application of patterns in models capturing real processes, simulating processes to predict levels of monitored variables and closing the loop with constant monitoring to adjust predictions.

10.3.2.1 Extensions for Matching and Application of Patterns

Behaviour-based Pattern Instances Verification. Assuming that patterns and models are available, the proposed pattern matching algorithms in Chapter 5 can be complemented in the context of the proposed framework with additional verification functionalities. The idea behind is to check if found pattern instances really *behave* as specified in their pattern descriptions. This is particularly interesting when working with executable process descriptions and patterns guiding the definition of new services. Considering that the identification of partial instances of process patterns can suggest the replacement of those partial instances by complete instances and their associated service-based implementations in processes, it would be interesting to verify if the complete pattern instances would *behave* as expected in the particular process models where they are instantiated. The focus would be on exploring if a process model can be safely transformed to incorporate a complete instance of a process pattern. One approach to check safety and adequate pattern instantiation is to analyse the ability of the transformed process model to be executed without deadlocks and according to the behaviour described in the pattern documentation.

One idea to check these properties on service-centric process models is to focus on the *interaction* perspective of services. This assumes that a participant interacting in a process is interested in other participants' behaviour. The boundary defined by a process pattern instance in a process model can be seen as a boundary defining the process performed by a special participant – so-called *pattern performer*. The pattern performer would be in charge of the execution of the pattern instance. In concrete models, this pattern performer would represent the set of participants executing the process steps abstracted by the process pattern configuration and implemented in the process pattern instance. It is expected for the pattern performer to simulate the behaviour indicated in its related process pattern configuration.

For executable process models, a process pattern can identify a process-centric service description. Because process models can identify an infinite set of possible states, to verify if the behaviour of a service is safe, it would be convenient to have a finite representation. The concept of an operating guideline (og) in [Lohmann 2007] can be used for these purposes. The behaviour of compatible process-centric services to a given service s can be captured in the operation guideline $og(s)$ associated to the service s. An og is represented by a deterministic annotated automaton, which

simulates all services $S = \{s_1, ..., s_n\}$ that can interact safely with s. Suppose a generic service s_P is derived from a process pattern P, and the operating guideline og_P identifies a set of compatible services S_P to the service s_P. A set of instances $\{PI\}$ of P in a process model would describe a set of services - say $\{s_{PI}\}$ - which are variations of the generic service s_P. It is expected for any service variation s_{PIx} in $\{s_{PI}\}$ to identify a set of compatible services S_{PIx} such that any participant safely interacting with s_P would also interact safely with s_{PIx}. Since operating guidelines can be represented as a graphs, to check the latter, a graph matching algorithm could identify if $og_P \subseteq og_{PIx}$, indicating that the interactive perspective of a pattern instance PIx - an its associated service variation S_{PIx} - can simulate the behaviour indicated by the generic pattern configuration P.

Exploration of the ideas in [Aalst 2010], [Aalst 2009b] and previous work in [Massuthe 2005], [Lohmann 2007], [Massuthe 2007] are the basis for future work to verify pattern instances used to guide the definition of new services.

Pattern-based Model Augmentation and Transformation. Business model and service architecture augmentation, and business model to service architecture transformation were suggested as part of the pattern-based techniques in the proposed framework (Section 3.3). To implement these techniques, concrete tools for horizonal and vertical *model transformations* are central. There is a significant amount of work for graph-based model transformations that can be adapted for processes and process-centric service architectures. For vertical transformation, direct translations from business process models to service descriptions are not straightforward, and existing errors – from the execution point of view – in business process models should be solved first. Tool support allowing end user interaction may be needed [Koehler 2008a]. In [Gacitua-Decar 2009b], transformation templates containing pattern dependant transformations are proposed to control and ensure safe transformation from business process model levels to software (service-centric) architecture levels. The idea is to provide transformation templates that are consistent with business level patterns and associated lower-level service architecture configurations. A similar approach is described in [Baresi 2006], but extended to a transformation from the entire (platform independent) business level architecture to a service-based architecture. The advantages of using a template transformation approach is that they can be reused across models. Patterns identified in process models can be directly transformed into sections of a service architecture without requiring the development of an specific transformation. However, an incremental integration of individual pattern-centric transformations should take place. Exploration of how transformation templates can be combined to create an integrated result expressed in a coherent service architecture can be a matter of future work. Ideas from pat-

tern and styles combination, pattern languages [Pahl 2009a], [Buschmann 2007] and combination of model transformations [Kleppe 2006], [Koehler 2008c] can be investigated and adapted to transformation template combination.

Semantic Similarity Between Graph Vertices. The proposed pattern matching techniques involve two main steps, vertex matching and iterative expansion steps. Vertex matching can involve complex semantic similarity calculations. In this work, one approach that involved natural language processing was presented. There are a number of natural language processing techniques that can be investigated in relation to their feasibility in process models. Process elements contain descriptions that include labels consisting of short sentences and possibly acronyms and abbreviations. On top of that, these sentences use concepts that are specific to determined domains. The use of a framework to experiment with ontology learning techniques [Gacitua 2008] to study different techniques for vertex similarity and abstraction extraction are considered as part of future work.

Process Model Generator. The evaluation of the proposed process pattern matching and discovery techniques or any other technique for process model analysis would benefit from advanced tools to generate experimental data emulating real scenarios of process models. A solution to generate graphs whose structure resembles aspects of the structure of process models is suggested in Section 8.7.3. Further development of a process graph generator could assist in the creation of a public repository of specialised graphs used to benchmark process model analysis techniques. The generator should have a set of parameters that allow the adjustment of structural characteristics of process models, a visualisation environment, functionalities for statistical analysis and the possibility to incorporate different process model analysis techniques. Some advances have been provided for architectural models such in [Varro 2005] and process mining techniques in [Li 2009], [ProM 2009].

10.3.2.2 Dynamic Pattern Matching and Discovery

Clusters detection and spread of information. Matching and discovery of patterns in this work was mainly addressed for *static* process models. A dynamic environment would benefit from additional aspects considering run-time changes of models (graphs). New techniques for dynamic pattern matching can take advantage of both the central ideas proposed for graph-based static pattern matching in this work and concepts borrowed from spread activation methods in information retrieval [Crestani 1997], [Cohen 1987], [Faloutsos 1995]. The idea behind it is that newly found elements to complete a match are identified using information that

is related to already matched elements in partial pattern instances. The relations between elements in models and their relevance, including those elements already matched, are dynamic and determined at run time.

Some organisations may not be willing to invest in analysis and documentation efforts to built a repository with patterns capturing best practices or proven and successful process-centric designs across projects. However, they can be interested in monitoring their processes, for instance, to improve key performance indicators in business operations. Process monitoring tools can report when key performance indicators in processes are at a desired level, however it is very difficult to determine what participants of the process, doing what actions, influencing whom and how, are the critical factors that take the process to a desired performance level or other quality measurements. The identification of critical configurations in processes that can lead to desired levels of performance can be seen as a problem of pattern identification. There are studies for detecting clusters (or communities) and for studying diffusion in networks [Girvan 2002], [Clauset 2004], [Jure 2010], [Gomez-Rodriguez 2010] that can be adapted to this scenario of pattern identification.

Community detection in networks considers the communities (or clusters) as groups of nodes with more and/or better interactions amongst its members than between its members and the remainder of the network. Approaches studying diffusion in networks attempt to observe the underlying network over which diffusions and propagations spread. These kind of studies could provide clues to identify the configurations in processes that could lead to desired levels of performance and the definition of process-level patterns. Network clusters often present characteristics of high cohesion and loose coupling [Dongen 2000], which are desired for (process-centric) services. Studies on diffusion and spreading of information can be an interesting model to be applied to the identification of critical roles in process patterns dynamically created in constantly evolving processes. These patterns could help to re-structure processes and services in adaptable process-centric service architectures. In [Jure 2010], the authors focus on the study of information diffusion and virus propagation. They provide techniques to trace the paths of diffusion and influence through networks and inferring the networks over which contagions propagate. Given the times when nodes of the network adopt pieces of information or become infected, they identify the optimal network that best explains the observed infection times. In a similar fashion, an optimal configuration of process elements could best explain the dynamics of a desired process phenomenon captured in a process pattern.

Graph clustering. The proposed solutions for process pattern matching can take a process model and a process pattern and identify the instances of the process

pattern in the process model. Instead, pattern discovery addresses the problem of finding frequently occurring substructures on large-scale process models. A pattern matching-based solution to pattern discovery was proposed in Chapter 6. Alternative graph clustering approaches proposed in other fields can be used to target improved performance – e.g., [Dongen 2000], [Kuramochi 2005], [Ketkar 2005], [Schaeffer 2007] in a graph-transaction setting. Such a scenario can become relevant when the interest is on finding a frequent pattern in a set of graph transactions. These graphs can refer, for example, to recorded modifications in models, events or transactions in processes and dynamically created service compositions.

10.3.2.3 Process Simulation and Monitoring

The interviews conducted to complement the evaluation of this work provided ideas of future work that can result in a practical impact. Process improvement and compliance to process regulations are two relevant activities related to process analysis and design. When analysing processes and considering the application of patterns that capture best practices or process regulations, analysts are interested in knowing the effect of applying the patterns before their implementation. Process simulation would be central for this task. Prior to that, actual processes can monitored to understand and register the values of the main variables, inputs and outputs to the processes. Monitoring requires adequate instrumentation in executable processes or software supporting operation of organisations. Process simulation involving a process model and real values of the main variables associated to the actual process and its inputs/outputs provide an *as-is* view of an actual process. Applying a pattern into a process model generates a modified process model. Simulating the modified model with the same variables and inputs/outputs of the original process can be exploited to provide predictions for the behaviour of the process before the implementation of modifications. After actually applying a pattern that modifies the real process, process monitoring can provide new information to confirm the predictions or to adjust the model. Process monitoring and model discovery has been recently considered by research groups, commercial spin-offs, and companies in the IT sector such as [ProM 2009], [Futura 2010] [Fujitsu 2008], giving promising opportunities to apply research associated to pattern-based analysis with monitoring and simulation in practice.

Bibliography

[Aalst 2003] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski and A. P. Barros. *Workflow Patterns*. Distributed and Parallel Databases, vol. 14, no. 1, pages 5–51, 2003. 19

[Aalst 2006] W. M. P. van der Aalst, A. de Medeiros and A. Weijters. *Process Equivalence: Comparing Two Process Models Based on Observed Behavior*. In Business Process Management, pages 129–144. 2006. 30

[Aalst 2007] W. M. P. van der Aalst, H. A. Reijers, A. J. M. M. Weijters, B. F. van Dongen, A. K. Alves de Medeiros, M. Song and H. M. W. Verbeek. *Business process mining: An industrial application*. Inf. Syst., vol. 32, no. 5, pages 713–732, 2007. 30, 147, 239

[Aalst 2009a] W. M. P. van der Aalst. *Process-Aware Information Systems: Lessons to Be Learned from Process Mining*. In Transactions on Petri Nets and Other Models of Concurrency II, volume 5460 of *Computer Science*, pages 1–26. Springer Berlin - Heidelberg, 2009. 239

[Aalst 2009b] W. M. P. van der Aalst, Arjan Mooij, Christian Stahl and Karsten Wolf. *Service Interaction: Patterns, Formalization, and Analysis*. In Formal Methods for Web Services, volume 5569, pages 42–88. Springer, 2009. 54, 242

[Aalst 2010] W. M.P. van der Aalst, N. Lohmann, P. Massuthe, C. Stahl and K. Wolf. *Multiparty Contracts: Agreeing and Implementing Interorganizational Processes*. The Computer Journal, vol. 53, pages 90–106, 2010. 242

[Agrawal 1994] Rakesh Agrawal and Ramakrishnan Srikant. *Fast Algorithms for Mining Association Rules in Large Databases*. In Proceedings of the 20th International Conference on Very Large Data Bases. Morgan Kaufmann Publishers Inc., 1994. 20, 39

[Aguilar-Saven 2004] R. S. Ruth Sara Aguilar-Saven. *Business process modelling: Review and framework*. International Journal of Production Economics, vol. 90, no. 2, pages 129–149, 2004. 73, 137, 237

[Aizenbud-Reshef 2006] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin and Y. Shaham-Gafni. *Model traceability*. IBM Syst. J., vol. 45, no. 3, pages 515–526, 2006. 69

[Aizenbud-Reshef 2007] Netta Aizenbud-Reshef, Ksenya Kveler and Inbal Ronen. *Service Identification using Combined Structured and Unstructured Legacy Code Analysis*. In IBM Programming Languages and Development Environments Seminar, IBM Haifa Labs, Israel, 2007. 26

[Aizenbud-Reshef 2009] Netta Aizenbud-Reshef, Ksenya Kveler and Inbal Ronen. *Service Identification in Legacy Source Code Using Structured and Unstructured Analyses (Patent Application Number 20090222429)*, 2009. 26

[Albani 2006] A. Albani, J. Dietz and J. Zaha. *Identifying Business Components on the Basis of an Enterprise Ontology*. In D. Konstantas, J-P. Bourrieres, M. Leonard and N. Boudjlida, editors, Interoperability of Enterprise Software and Applications, pages 335–347. Springer, 2006. 26, 129

[Alexander 1977] Christopher Alexander, Sara Ishikawa and Murray Silverstein. A pattern language: towns, buildings, construction. Oxford University Press, 1977. 16

[Allen 1997] Robert Allen and David Garlan. *A formal basis for architectural connection*. ACM Trans. Softw. Eng. Methodol., vol. 6, no. 3, pages 213–249, 1997. 53

[Alonso 2004] G. Alonso, F. Casati, H. Kuno and V. Machiraju. Web services: Concepts, architectures and applications. Springer Verlag, 2004. 53

[Altova 2008] Altova. *MapForce - Graphical Data Mapping, Conversion, and Integration Tool*, 2008. Available from: `http://www.altova.com/mapforce.html`. 175

[Ambler 1998] Scott W. Ambler. Process patterns: Building large scale systems using object technology, volume 15 of *Managing Object Technology*. Cambridge Univ. Press, 1998. 19

[Arsanjani 2004] Ali Arsanjani. *Service-oriented modeling and architecture*, 2004. Available from: `http://www-128.ibm.com/developerworks/webservices/library/ws-soa-design1/`. 13, 14, 237

[Avgeriou 2005] P. Avgeriou and U. Zdun. *Architectural patterns revisited - a pattern language*. In 10th European Conference on Pattern Languages of Programs (EuroPlop), pages 1–39, Irsee, Germany, 2005. 17, 19

[Awad 2008a] Ahmed Awad, Gero Decker and Mathias Weske. *Efficient Compliance Checking Using BPMN-Q and Temporal Logic*. In Business Process Management, volume 5240, pages 326–341. Springer, 2008. 32

[Awad 2008b] Ahmed Awad, Artem Polyvyanyy and Mathias Weske. *Semantic Querying of Business Process Models*. In 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC'08), pages 85–94. IEEE Computer Society, 2008. 32, 125

[Awad 2008c] Ahmed Awad and Frank Puhlmann. *Structural Detection of Deadlocks in Business Process Models*. In Business Information Systems, volume 7 of *Lecture Notes in Business Information Processing*, pages 239–250. Springer, 2008. 212

[Awad 2009] Ahmed Awad, Matthias Weidlich and Mathias Weske. *Specification, Verification and Explanation of Violation for Data Aware Compliance Rules*. In 7th International Conference on Service Oriented Computing, ICSOC-SERVICE WAVE'09, volume 5900 of *LNCS*, pages 500–515. Springer, 2009. 225

[Awad 2010] Ahmed Awad and Mathias Weske. *Visualization of Compliance Violation in Business Process Models*. In Business Process Management Workshops. Part 2, volume 43 of *LNBIP*, pages 182–193. Springer, 2010. 225

[Babar 2004] M. A. Babar. *Scenarios, quality attributes, and patterns: capturing and using their synergistic relationships for product line architectures*. In 11th Asia-Pacific Software Engineering Conference (APSEC'04), pages 574–578. IEEE, 2004. 59, 62

[Bae 2006] Joonsoo Bae, Ling Liu, James Caverlee and William B. Rouse. *Process Mining, Discovery, and Integration using Distance Measures*. In Proceedings of the IEEE International Conference on Web Services (ICWS'06), pages 479–488. IEEE Computer Society, 2006. 29, 147

[Baresi 2006] Luciano Baresi, Reiko Heckel, Sebastian Thone and Daniel Varro. *Style-based modeling and refinement of service-oriented architectures*. Software and Systems Modeling, vol. 5, no. 2, pages 187–207, 2006. 37, 242

[Barros 2007] Oscar Barros. *Business process patterns and frameworks: Reusing knowledge in process innovation*. Business Process Management Journal, vol. 13, no. 1, pages 47 – 69, 2007. 19, 20, 54, 143, 239

[Basili 1996] V. R. Basili. *The role of experimentation in software engineering: past, current, and future*. In 18th International Conference on Software Engineering (ICSE'96), pages 442–449. IEEE, 1996. 180

[Bass 2004] Len Bass, Paul Clements and Rick Kazman. Software architecture in practice. Addison-Wesley Professional, second édition, 2004. 51

[Beaver 2005] J. M. Beaver, G. A. Schiavone and J. S. Berrios. *Predicting software suitability using a Bayesian belief network*. In Fourth International Conference on Machine Learning and Applications, pages 82–88. IEEE, 2005. 280

[Beeri 2008] Catriel Beeri, Anat Eyal, Simon Kamenkovich and Tova Milo. *Querying business processes with BP-QL*. Information Systems, vol. 33, no. 6, pages 477–507, 2008. 29, 31

[Bengtsson 2004] PerOlof Bengtsson, Nico Lassing, Jan Bosch and Hans van Vliet. *Architecture-level modifiability analysis (ALMA)*. Journal of Systems and Software, vol. 69, no. 1-2, pages 129–147, 2004. 141, 143, 144, 146, 147, 161

[Bernus 2003] P. Bernus, L. Nemes and G. Schmidt. Handbook on enterprise architecture. International Handbooks on Information Systems. Springer, 2003. 24

[Bhatti 2005] Shahid Nazir Bhatti. *Why quality?': ISO 9126 software quality metrics (Functionality) support by UML suite*. SIGSOFT Softw. Eng. Notes, vol. 30, no. 2, pages 1–5, 2005. 280

[Blondel 2004] Vincent D. Blondel, Anahi Gajardo, Maureen Heymans, Pierre Senellart and Paul Van Dooren. *A Measure of Similarity between Graph Vertices: Applications to Synonym Extraction and Web Searching*. SIAM Review, vol. 46, no. 4, pages 647–666, 2004. 97

[Bottoni 2009] Paolo Bottoni, Esther Guerra and Juan de Lara. *Formal Foundation for Pattern-Based Modelling*. In Fundamental Approaches to Software Engineering (FASE'09), volume 5503 of *LNCS*, pages 278–293. Springer, 2009. 56

[Brahe 2007] Steen Brahe. *BPM on Top of SOA: Experiences from the Financial Industry*. In Business Process Management, volume 4714, pages 96–111. Springer, 2007. 12

[Brian 2006] Berenbach Brian and Borotto Gail. *Metrics for model driven requirements development*. In 28th international conference on Software engineering (ICSE'06), pages 445–451, Shanghai, China, 2006. ACM. 280

[Briand 2006] L. C. Briand, Y. Labiche, L. O Sullivan and M. M. Sowka. *Automated impact analysis of UML models*. Journal of Systems and Software, vol. 79, no. 3, pages 339–352, 2006. 67

[Bringmann 2008] Bjorn Bringmann and Siegfried Nijssen. *What Is Frequent in a Single Graph?'*. In Advances in Knowledge Discovery and Data Mining, volume 5012 of *LNCS*, pages 858–863. Springer, 2008. 39, 137

[Bunke 1997] H. Bunke. *On a relation between graph edit distance and maximum common subgraph*. Pattern Recognition Letters, vol. 18, no. 8, pages 689–694, 1997. 31, 36, 237

[Bunke 2005] Horst Bunke, Christophe Irniger and Michel Neuhaus. *Graph Matching - Challenges and Potential Solutions*. In Image Analysis and Processing - ICIAP'05, volume 3617 of *LNCS*, pages 1–10. Springer, 2005. 34, 35, 92

[Buschmann 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal. Pattern-oriented software architecture: a system of patterns. John Wiley and Sons, Inc. New York, NY, USA, 1996. 17

[Buschmann 2003] Frank Buschmann and Kevlin Henney. *Beyond the Gang of Four. Tutorial at OOPSLA'03*, 2003. Available from: `http://www.two-sdg.demon.co.uk/curbralan/papers/jaoo/BeyondTheGangOfFour.pdf`. 57

[Buschmann 2007] Frank Buschmann, Kevlin Henney and Douglas C. Schmidt. Pattern-oriented software architecture: On patterns and pattern languages. Wiley and Sons, 2007. 17, 22, 55, 56, 63, 143, 144, 243, 288

[Cardoso 2006] J. Cardoso, J. Mendling, G. Neumann and H. Reijers. *A Discourse on Complexity of Process Models*. In Business Process Management Workshops, volume 4103, pages 117–128. Springer, 2006. 286

[Cha 2007] Sung-Hyuk Cha. *Comprehensive Survey on Distance-Similarity Measures between Probability Density Functions*. International Journal of Mathematical Models and Methods in Applied Sciences, vol. 1, no. 1, pages 300–307, 2007. 124

[Chen 2005] Hong-Mei Chen, Rick Kazman and Aditya Garg. *BITAM: An engineering-principled method for managing misalignments between business and IT architectures*. Science of Computer Programming, vol. 57, no. 1, pages 5–26, 2005. 143

[Clauset 2004] A. Clauset, M. E. J. Newman and C. Moore. *Finding community structure in very large networks*. Physical Review E (Statistical, Nonlinear, and Soft Matter Physics), vol. 70, no. 6, pages 66111–1, 2004. 244

[Cohen 1987] Paul R. Cohen and Rick Kjeldsen. *Information retrieval by constrained spreading activation in semantic networks*. Information Processing & Management, vol. 23, no. 4, pages 255–268, 1987. 138, 243

[Coleman 1994] Don Coleman, Dan Ash, Bruce Lowther and Paul Oman. *Using Metrics to Evaluate Software System Maintainability*. Computer, vol. 27, no. 8, pages 44–49, 1994. 282

[Conte 2004] D. Conte, P. Foggia, C. Sansone and M. Vento. *Thirty Years of Graph Matching in Pattern Recognition*. International Journal of Pattern Recognition & Artificial Intelligence, vol. 18, no. 3, pages 265–298, 2004. 34, 92, 93, 97, 98, 109

[Cook 1995] Jonathan E. Cook and Alexander L. Wolf. *Automating Process Discovery through Event-Data Analysis*. In 17th International Conference on Software Engineering ICSE'95, pages 73–82. ACM, 1995. 40

[Corradini 1996] A. Corradini, U. Montanari and F. Rossi. *Graph processes*. Fundam. Inf., vol. 26, no. 3-4, pages 241–265, 1996. 73, 237, 269, 272

[Corrales 2006] Juan Corrales, Daniela Grigori and Mokrane Bouzeghoub. *BPEL Processes Matchmaking for Service Discovery*. In On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, pages 237–254. 2006. 27, 28, 31

[Corrales 2008] J. C. Corrales, D. Grigori, M. Bouzeghoub and J. E. Burbano. *BeMatch: a platform for matchmaking service behavior models*. In 11th Int. Conf. on Extending Database Technology: Advances in Database Technology, volume 261, pages 695–699, Nantes, France, 2008. ACM. 27

[Crestani 1997] F. Crestani. *Application of Spreading Activation Techniques in Information Retrieval*. Artificial Intelligence Review, vol. 11, no. 6, pages 453–482, 1997. 138, 243

[Curry 2008] E. Curry and P. Grace. *Flexible Self-Management Using the Model-View-Controller Pattern*. Software, IEEE, vol. 25, no. 3, pages 84–90, 2008. 18

[Daniel 2009] F. Daniel, F. Casati, V. D'Andrea, S. Strauch, D. Schumm, F. Leymann, E. Mulo, U. Zdun, S. Dustdar, A. Sebahi, F. de Marchi and M. Hacid. *Business Compliance Governance in Service-Oriented Architectures*. In 23th Int. Conf. on Advanced Information Networking and Applications (AINA'09), pages 113–120, Bradford, United Kingdom, 2009. IEEE. 140, 144, 159, 282, 288

[De Antonellis 2003] V. De Antonellis, M. Melchiori and P. Plebani. *An approach to Web Service compatibility in cooperative processes*. In Symposium on Applications and the Internet Workshops, pages 95–100. IEEE, 2003. 29

[de Lara 2007] Juan de Lara, Roswitha Bardohl, Hartmut Ehrig, Karsten Ehrig, Ulrike Prange and Gabriele Taentzer. *Attributed graph transformation with node type inheritance*. Theoretical Computer Science, vol. 376, no. 3, pages 139–163, 2007. 76

[Diestel 2005] R. Diestel. Graph theory. Springer, 2005. 82

[Dijkman 2004] R.M. Dijkman and M. Dumas. *Service-oriented Design: A Multi-viewpoint Approach*. International Journal of Cooperative Information Systems (IJCIS). Special Issue on Service Oriented Modeling, vol. 13, no. 4, pages 337–368, 2004. 24

[Dijkman 2009a] Remco Dijkman, Marlon Dumas and Luciano Garcia-Banuelos. *Graph Matching Algorithms for Business Process Model Similarity Search*. In Business Process Management, volume 5701 of *LNCS*, pages 48–63. Springer, 2009. 30, 31

[Dijkman 2009b] R.M. Dijkman, B.F. van Dongen, M. Dumas, R. Kaarik and J. Mendling. *Similarity of Business Process Models: Metrics and Evaluation*. Technical report, Eindhoven University of Technology, 2009. 30, 31, 125

[Dong 2007] Jing Dong, Yajing Zhao and Tu Peng. *Architecture and Design Pattern Discovery Techniques - A Review*. In International Conference on Software Engineering Research and Practice (SERP), pages 621–627, USA, 2007. 24

[Dongen 2000] Stijn van Dongen. *Graph Clustering by Flow Simulation*. PhD thesis, University of Utrecht, 2000. 244, 245

[Dongen 2008] Boudewijn F. van Dongen, Remco Dijkman and Jan Mendling. *Measuring Similarity between Business Process Models*. In Advanced Information Systems Engineering, volume 5240, pages 450–464. Springer, 2008. 30, 31, 125

[Dorr 1995a] Heiko Dorr. *The abstract machine for graph rewriting - supporting a fast implementation*. In Efficient Graph Rewriting and Its Implementation, volume 922 of *LNCS*, pages 123–162. Springer, 1995. 39

[Dorr 1995b] Heiko Dorr. *UBS-Graph rewriting systems - matching subgraphs in constant time*. In Efficient Graph Rewriting and Its Implementation, volume 922 of *LNCS*, pages 35–89. Springer, 1995. 39

[Dustdar 2005] Schahram Dustdar and Wolfgang Schreiner. *A Survey on Web Services Composition*. International Journal of Web and Grid Services, vol. 1, no. 1, pages 1–30, 2005. 27

[Ehrig 1997] H. Ehrig, R. Heckel, M. Korff, M. Lowe, L. Ribeiro, A. Wagner and A. Corradini. *Algebraic approaches to graph transformation. Part II: single pushout approach and comparison with double pushout approach*. In Grzegorz Rozenberg, editor, Handbook of graph grammars and computing by graph transformation, volume I. foundations, pages 247 – 312. World Scientific Publishing Co., Inc., 1997. 73, 276

[Ehrig 1999a] H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, editors. Handbook of graph grammars and computing by graph transformation, volume 2: Applications, languages and tools. World Scientific, 1999. 269

[Ehrig 1999b] H. Ehrig, M. Gajewsky and F. Parisi-Presicce. Handbook of graph grammars and computing by graph transformation, volume 3: Concurrency, parallelism, and distribution. World Scientific, 1999. 32

[Ehrig 2004] Hartmut Ehrig, Ulrike Prange and Gabriele Taentzer. *Fundamental Theory for Typed Attributed Graph Transformation*. In Graph Transformations, volume 3256, pages 161–177. Springer, 2004. 269

[Ehrig 2006a] H. Ehrig, K. Ehrig, U. Prange and G. Taentzer. Fundamentals of algebraic graph transformation. Monographs in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2006. 37, 89, 275

[Ehrig 2006b] Hartmut Ehrig and Karsten Ehrig. *Overview of Formal Concepts for Model Transformations Based on Typed Attributed Graph Transformation*. ENTCS, vol. 152, pages 3–22, 2006. 76, 269, 275

[Ehrig 2007] Marc Ehrig, Agnes Koschmider and Andreas Oberweis. *Measuring Similarity between Semantic Business Process Models*. In John F. Roddick and Annika Hinze, editors, 4th Asia-Pacific Conf. on Conceptual Modelling (APCCM'07), volume 67 of *CRPIT*, pages 71–80, Ballarat, Australia, 2007. ACS. 31

[Ehrig 2008] H. Ehrig, K. Ehrig, C. Ermel and U. Prange. *Generalized Typed Attributed Graph Transformation Systems based on Morphisms Changing Type Graphs and Data Signatures*. Technical report, TU Berlin, 2008. 74

[EPN 2010] EPN and NACHA, 2010. Available from: `http://www.electronicpayments.org/`. 59

[Eriksson 1998] Hans-Erik Eriksson and Magnus Penker. Business modeling with uml: Business patterns at work. John Wiley and Sons, Inc., 1998. 19

[Erl 2004] Thomas Erl. Service-oriented architecture: Concepts, technology, and design. Prentice Hall, 2004. 13, 25, 49, 93, 129, 237, 284

[Erl 2008] Thomas Erl. Soa design patterns. Prentice Hall - Pearson, 2008. 65, 66, 151, 171, 174, 299

[Ermel 1999] C. Ermel, M. Rudolf and G. Taentzer. *The AGG approach: language and environment*. In Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools, pages 551–603. World Scientific Publishing Co., Inc., 1999. 38

[Erradi 2006] A. Erradi, S. Anand and N. Kulkarni. *Evaluation of Strategies for Integrating Legacy Applications as Services in a Service Oriented Architecture*. In A. Sriram, editor, IEEE Int. Conf. on Services Computing (SCC'06)., pages 257–260. IEEE, 2006. 12

[Eshuis 2007] Rik Eshuis and Paul Grefen. *Structural Matching of BPEL Processes*. In W. Zimmermann, B. Koenig-Ries and C. Pahl, editors, Proceedings of the Fifth European Conference on Web Services, pages 171–180, Halle, Germany, 2007. IEEE Computer Society. 28

[Fagan 1976] M. E. Fagan. *Design and code inspections to reduce errors in program development*. IBM Syst. J., vol. 15, no. 3, pages 182–211, 1976. 19

[Fahmy 2000] H. Fahmy and R. C. Holt. *Software architecture transformations*. In International Conference on Software Maintenance (ICSM'00), pages 88–96. IEEE Computer Society, 2000. 82, 120

[Faloutsos 1995] Christos Faloutsos and Douglas W. Oard. *A survey of Information Retrieval and Filtering Methods. CS-TR-3514.* Technical report, University of Maryland, 1995. 138, 243

[Fernandez 2001] Mirtha-Lina Fernandez and Gabriel Valiente. *A graph distance metric combining maximum common subgraph and minimum common supergraph*. Pattern Recognition Letters, vol. 22, no. 6-7, pages 753–758, 2001. 36

[Fiala 2005] Jiri Fiala and Daniel Paulusma. *A complete complexity classification of the role assignment problem*. Theoretical Computer Science, vol. 349, no. 1, pages 67–81, 2005. 271

[Fiala 2007] Jiri Fiala. *Structure And Complexity of Locally Constrained Graph Homomorphisms*. PhD thesis, Charles University, Faculty of Mathematics And Physics, 2007. 269, 272

[Fiala 2008] Jiri Fiala and Jan Kratochvil. *Locally constrained graph homomorphisms–structure, complexity, and applications*. Computer Science Review, vol. 2, no. 2, pages 97–111, 2008. 95, 108, 271

[Forgy 1982] Charles L. Forgy. *Rete: A fast algorithm for the many pattern-many object pattern match problem*. Artificial Intelligence, vol. 19, no. 1, pages 17–37, 1982. 36

[Fujaba 2010] Fujaba. *Fujaba Tool Suite*, 2010. Available from: `http://www.fujaba.de`. 37

[Fujitsu 2008] Fujitsu. *Interstage BPM - Business Process Discovery and Visualisation*, 2008. Available from: `http://www.fujitsu.com/global/services/software/interstage/bpm/apd.html`. 245

[Futura 2010] Futura. *Futura Process Intelligence*, 2010. Available from: `http://www.futuratech.nl`. 245

[Gacitua-Decar 2008a] Veronica Gacitua-Decar and Claus Pahl. *Business model driven Service Architecture Design for Enterprise Application Integration*. In Int. Conf. on Business Innovation and Information Technology. ICBIIT'08, pages 74–85, Dublin, Ireland, 2008. Logos Verlag. 171

[Gacitua-Decar 2008b] Veronica Gacitua-Decar and Claus Pahl. *Towards Pattern-Based Service Identification*. In W. Binder and S. Dustdar, editors, 3rd Workshop on Emerging Web Services Technology - WEWST'08., pages 15–30, Dublin, Ireland, 2008. 103

[Gacitua-Decar 2009a] Veronica Gacitua-Decar and Claus Pahl. *Automatic Business Process Pattern Matching for Enterprise Services Design*. In IEEE World Conference on Services - II. SERVICES -2'09, pages 111–118. IEEE, 2009. 70, 103, 181, 198

[Gacitua-Decar 2009b] Veronica Gacitua-Decar and Claus Pahl. *Ontology-based Patterns for the Integration of Business Processes and Enterprise Application Architectures*. In G. Mentzas, T. Bouras, P. Gouvas and A. Friesen, editors, Semantic Enterprise Application Integration. IGI Publishers, Ltd., 2009. 52, 242

[Gacitua-Decar 2009c] Veronica Gacitua-Decar and Claus Pahl. *Towards Reuse of Business Processes Patterns to Design Services*. In W. Binder and S. Dustdar, editors, Emerging Web Services Technology, volume III of *Whitestein Series in Software Agent Technologies and Autonomic Computing*, pages 15–36. Springer - Birkhauser, 2009. 64, 70, 74

[Gacitua 2008] Ricardo Gacitua, Pete Sawyer and Paul Rayson. *A flexible framework to experiment with ontology learning techniques*. Knowledge-Based Systems, vol. 21, no. 3, pages 192–199, 2008. 211, 243

[Gallagher 2006a] Brian Gallagher. *Matching structure and semantics: A survey on graph-based pattern matching*. In AAAI Fall Symposium on Capturing and Using Patterns for Evidence Detection (AAAI FS'06), pages 45–53, Washington, DC, 2006. AAAI Pres. 34, 35

[Gallagher 2006b] Brian Gallagher. *The State of the Art in Graph-Based Pattern Matching*. Technical report, Center for Applied Scientific Computing. Lawrence Livermore National Laboratory, 2006. 34, 98, 109

[Gamma 1993] Erich Gamma, Richard Helm, Ralph E. Johnson and John M. Vlissides. *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. In Oscar M. Nierstrasz, editor, 7th European Conference on Object-Oriented Programming, volume 707 of *LNCS*, pages 406 – 431, Kaiserslautern, Germany, 1993. Springer. 22, 143

[Gamma 1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design patterns: Elements of reusable object-oriented software. Addison-Wesley Professional Computing Series, 1995. ix, 16, 17, 79, 80, 151

[Gardner 2003] Tracy Gardner. *UML Modelling of Automated Business Processes with a Mapping to BPEL4WS*. In First European Workshop on Web Services and Object Orientation (EOOWS03), in conjunction with ECOOP03, Darmstadt, Germany, 2003. 14, 169, 174

[Garlan 2006] David Garlan and Bradley Schmerl. *Architecture-driven Modelling and Analysis*. In Tony Cant, editor, SCS06, volume 248 of *ACM International Conference Proceeding Series*, pages 3–17. Australian Computer Society, Inc., Melbourne, Australia, 2006. 53, 58

[Ghose 2008] Aditya Ghose and George Koliadis. *Auditing Business Process Compliance*. In Service-Oriented Computing - ICSOC'07, volume 4749, pages 169–180. Springer, 2008. 144, 159, 225, 282

[Giesecke 2007] Simon Giesecke, Wilhelm Hasselbring and Matthias Riebisch. *Classifying architectural constraints as a basis for software quality assessment*. Adv. Eng. Inform., vol. 21, no. 2, pages 169–179, 2007. 18, 19

[Girvan 2002] M. Girvan and M. E. J. Newman. *Community structure in social and biological networks*. Proceedings of the National Academy of Sciences of the United States of America, vol. 99, no. 12, pages 7821–7826, 2002. 244

[Glenn 1988] E. Krasner Glenn and T. Pope Stephen. *A cookbook for using the model-view controller user interface paradigm in Smalltalk-80*. J. Object Oriented Program., vol. 1, no. 3, pages 26–49, 1988. 18

[Golani 2003] Mati Golani and Shlomit Pinter. *Generating a Process Model from a Process Audit Log*. In Business Process Management, pages 1020–1020. 2003. 39, 137, 181, 184

[Gomes 2003] Maria Cecilia Gomes, Omer F. Rana and Jose C. Cunha. *Pattern operators for grid environments*. Sci. Program., vol. 11, no. 3, pages 237–261, 2003. 23

[Gomes 2008] M. C. Gomes, Omer Rana and J. C. Cunha. *Extending Grid-Based Workflow Tools with Patterns/Operators*. The International Journal of High Performance Computing Applications, vol. 22, no. 3, pages 301–318, 2008. 23

[Gomez-Rodriguez 2010] M. Gomez-Rodriguez, J. Leskovec and A. Krause. *Inferring Networks of Diffusion and Influence*. In Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'10), pages 1–10. ACM, 2010. 244

[Gorton 2004] I. Gorton and A. Liu. *Architectures and technologies for enterprise application integration*. In A. Liu, editor, 26th International Conference on Software Engineering. ICSE 2004., pages 726–727, 2004. 12

[GraphML 2010] Working Group GraphML. *The GraphML File Format*, 2010. Available from: `http://graphml.graphdrawing.org`. 211

[Greco 2005] Gianluigi Greco, Antonella Guzzo, Giuseppe Manco and Domenico Sacca. *Mining and Reasoning on Workflows*. IEEE Trans. on Knowl. and Data Eng., vol. 17, no. 4, pages 519–534, 2005. 39, 40, 138, 147

[Greco 2008] Gianluigi Greco, Antonella Guzzo and Luigi Pontieri. *Mining taxonomies of process models*. Data Knowl. Eng., vol. 67, no. 1, pages 74–102, 2008. 40

[Gregory 1995] D. Abowd Gregory, Allen Robert and Garlan David. *Formalizing style to understand descriptions of software architecture*. ACM Trans. Softw. Eng. Methodol., vol. 4, no. 4, pages 319–364, 1995. 18

[Gruhn 2006] Volker Gruhn and Ralf Laue. *Complexity Metrics for Business Process Models*. In W. Abramowicz and H.C. Mayr, editors, 9th International Conference on Business Information Systems (BIS'06), volume 85 of *LNI*, pages 1–12, Klagenfurt, Austria, 2006. 156, 287

[Gschwind 2008] Thomas Gschwind, Jana Koehler and Janette Wong. *Applying Patterns During Business Process Modeling*. In M. Dumas, M. Reichert and M.-C. Shan, editors, Business Process Management (BPM 2008), volume 5240, pages 4–19. Springer, 2008. 16, 19, 239

[Gucegliogl u 2005] A. Selcuk Gucegliogl u and Onur Demirors. *Using Software Quality Characteristics to Measure Business Process Quality*. In Business Process Management, volume 3649 of *LNCS*, pages 374–379. Springer, 2005. 280, 286

[Gunay 2007] Akin Gunay and Pinar Yolum. *Structural and Semantic Similarity Metrics for Web Service Matchmaking*. In E-Commerce and Web Technologies, volume 4655, pages 129–138. Springer, 2007. 31, 32

[Han 2007] Jiawei Han, Hong Cheng, Dong Xin and Xifeng Yan. *Frequent pattern mining: current status and future directions*. Data Mining and Knowledge Discovery, vol. 15, no. 1, pages 55–86, 2007. 39, 137

[Harrison 2007] Neil Harrison and Paris Avgeriou. *Leveraging Architecture Patterns to Satisfy Quality Attributes*. In Software Architecture, volume 4758 of *LNCS*, pages 263–270. Springer, 2007. 59, 62, 157

[Harrison 2008] N. B. Harrison and P. Avgeriou. *Analysis of Architecture Pattern Usage in Legacy System Architecture Documentation*. In Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA'08), pages 147–156. IEEE, 2008. 157

[Heckel 1995] Reiko Heckel and Annika Wagner. *Ensuring Consistency of Conditional Graph Grammars - A Constructive Approach*. ENTCS, vol. 2, pages 118–126, 1995. 88, 89, 277

[Heckel 2002] Reiko Heckel, Jochen Kuster and Gabriele Taentzer. *Confluence of Typed Attributed Graph Transformation Systems*. In Graph Transformation, volume 2505 of *LNCS*, pages 161–176. Springer Berlin Heidelberg, 2002. 269, 274

[Heckel 2006] R. Heckel. *Graph Transformation in a Nutshell*. ENTCS, vol. 148, pages 187–198, 2006. 37

[Hell 2004] P. Hell and J. Nesetril. *Graphs and Homomorphisms*. Oxford Lecture Series in Mathematics and Its Applications, vol. 28, 2004. xi, 269, 270, 272

[Henderson 1993] J. Henderson and N. Venkatraman. *Strategic alignment: Leveraging information technology for transforming organizations*. IBM Systems Journal, vol. 32, no. 1, pages 4–16, 1993. 143

[Hentrich 2006] C. Hentrich and U. Zdun. *Patterns for process-oriented integration in service-oriented architectures*. In 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006), pages 141–198, Irsee, Germany, 2006. 22

[Hevner 2004] A.R. Hevner, S.T. March, J. Park and S. Ram. *Design science in information systems research*. MIS Quarterly, vol. 28, no. 1, pages 75–105, 2004. 5

[Hevner 2010] Alan Hevner and Samir Chatterjee. Design science research in information systems: Theory and practice. Integrated Series in Information Systems. Springer, 2010. xiii, 5, 7

[Hill 2009] Janelle B. Hill, Michele Cantara, Marc Kerremans and Daryl C. Plummer. *Magic Quadrant for Business Process Management Suites*. Technical report, Gartner Inc., 2009. 144, 145, 239

[Hirzalla 2009] Mamoun Hirzalla, Jane Cleland-Huang and Ali Arsanjani. *A Metrics Suite for Evaluating Flexibility and Complexity in Service Oriented Architectures*. In Service-Oriented Computing Workshops (ICSOC'08), volume 5472, pages 41–52. Springer, 2009. 156, 287

[Hohpe 2004] G. Hohpe and B. Woolf. Enterprise integration patterns. Addison-Wesley Boston, 2004. 96, 140, 151, 153

[Huang 2004] Kui Huang, Zhaotao Zhou, Yanbo Han, Gang Li and Jing Wang. *An Algorithm for Calculating Process Similarity to Cluster Open-Source Process Designs*. In Grid and Cooperative Computing - GCC'04 Workshops, volume 3252 of *LNCS*, pages 107–114. Springer, 2004. 29

[Inokuchi 2005] Akihiro Inokuchi, Takashi Washio and Hiroshi Motoda. *A General Framework for Mining Frequent Subgraphs from Labeled Graphs*. Fundamenta Informaticae, vol. 66, no. 1/2, pages 53–82, 2005. 40, 132

[Intan 2002] Rolly Intan and Masao Mukaidono. *Degree of Similarity in Fuzzy Partition*. In Advances in Soft Computing, volume 2274 of *LNCS*, pages 99–107. Springer, 2002. 29

[Janssen 2005] Marijn Janssen and Anthony M. Cresswell. *An enterprise application integration methodology for e-government*. Journal of Enterprise Information Management, vol. 18, no. 5, pages 531–547, 2005. 169, 280, 287

[Johannesson 2001] Paul Johannesson and Erik Perjons. *Design principles for process modelling in enterprise application integration*. Information Systems, vol. 26, no. 3, pages 165–184, 2001. 12, 142

[Jon 1999] M. Kleinberg Jon. *Authoritative sources in a hyperlinked environment*. J. ACM, vol. 46, no. 5, pages 604–632, 1999. 23

[Jones 2005] Steve Jones. *A Methodology for Service Architectures*. Technical report, OASIS, 2005. 13

[Jung 2006] Jae-Yoon Jung and Joonsoo Bae. *Workflow Clustering Method Based on Process Similarity*. In Computational Science and Its Applications - ICCSA'06, volume 3981 of *LNCS*, pages 379–389. Springer, 2006. 33, 39, 137

[Jung 2008] Jae-Yoon Jung, Joonsoo Bae and Ling Liu. *Hierarchical Business Process Clustering*. In IEEE International Conference on Services Computing (SCC'08), volume 2, pages 613–616, Honolulu, HI, 2008. IEEE. 33

[Jurack 2009] Stefan Jurack, Leen Lambers, Katharina Mehner, Gabriele Taentzer and Gerd Wierse. *Object Flow Definition for Refined Activity Diagrams*. In Fundamental Approaches to Software Engineering (FASE), volume 5503 of *LNCS*, pages 49–63. Springer, 2009. 70

[Jure 2010] Leskovec Jure, J. Lang Kevin and Mahoney Michael. *Empirical comparison of algorithms for network community detection*. In 19th International Conference on World Wide Web, Raleigh, North Carolina, USA, 2010. ACM. 244

[Kampffmeyer 2007] Holger Kampffmeyer and Steffen Zschaler. *Finding the Pattern You Need: The Design Pattern Intent Ontology*. In Model Driven Engineering Languages and Systems, volume 4735, pages 211–225. Springer, 2007. 56

[Kavianpour 2009] Mansour Kavianpour. *SOA and Large Scale and Complex Enterprise Transformation*. In Service-Oriented Computing - ICSOC'07, volume 4749 of *LNCS*, pages 530–545. Springer, 2009. 25

[Kazman 1998] R. Kazman. *Assessing architectural complexity*. In Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering, pages 104–112, 1998. 156, 286, 287

[Kazman 2000] R. Kazman, M. Klein and P. Clements. Atam: Method for architecture evaluation (CMU/SEI-2000-TR-004). Software Engineering Institute (SEI), Carnegie Mellon University, 2000. 141, 142

[Ketkar 2005] N. S. Ketkar, L. B. Holder and D. J. Cook. *Subdue: Compression-based Frequent Pattern Discovery in Graph Data*. In 1st Int. Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations, pages 71–76, Chicago, IL, 2005. ACM. 245

[Kharbili 2008] M. El Kharbili, A. K. Alves de Medeiros, S. Stein and W. M. P. van der Aalst. *Business Process Compliance Checking: Current State and Future Challenges*. In Modellierung betrieblicher Informationssysteme (MobIs), volume 141 of *LNI*, pages 107–113, Saarbrucken, Germany, 2008. GI. 140, 144, 159, 225, 282

[Kim 2007] Dae-Kyoo Kim and Charbel El Khawand. *An approach to precisely specifying the problem domain of design patterns*. Journal of Visual Languages and Computing, vol. 18, no. 6, pages 560–591, 2007. 22, 62

[Kim 2008] Soon-Kyeong Kim and David Carrington. *A formalism to describe design patterns based on role concepts*. Formal Aspects of Computing, vol. 21, no. 5, pages 397–420, 2008. 56

[Klein 2004] M. Klein and A. Bernstein. *Toward high-precision service retrieval*. IEEE Internet Computing, vol. 8, no. 1, pages 30–36, 2004. 32

[Kleppe 2006] Anneke Kleppe. *Proceedings of the First European Workshop on Composition of Model Transformations - CMT'06. Technical Report TR-CTIT-06-34*. Technical report, Centre for Telematics and Information Technology, University of Twente, Enschede, 2006. 243

[Koehler 2006] J. Koehler, R. Hauser, J. Kuster, K. Ryndina, J. Vanhatalo and M. Wahler. *The Role of Visual Modeling and Model Transformations in Business driven Development*. In R. Bruni and D. Varro, editors, GT-VMT 2006, ENTCS, Vienna, Austria, 2006. 14, 169

[Koehler 2007] J. Koehler and J. Vanhatalo. *Process anti-patterns: How to avoid the common traps of business process modeling. Part 1-2*. IBM WebSphere Developer Technical Journal, vol. 10, no. 2-4, 2007. 171

[Koehler 2008a] J. Koehler, T. Gschwind, J. Kuster, C. Pautasso, K. Ryndina, J. Vanhatalo and H. Volzer. *Combining Quality Assurance and Model Transformations in Business-Driven Development*. In Int. Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance, volume 5088 of *LNCS*, pages 1–16. Springer, 2008. 242

[Koehler 2008b] Jana Koehler, Rainer Hauser, Jochen Kuster, Ksenia Ryndina, Jussi Vanhatalo and Michael Wahler. *The Role of Visual Modeling and Model Transformations in Business-driven Development*. ENTCS, vol. 211, pages 5–15, 2008. 12, 14, 64, 69

[Koehler 2008c] Jana Koehler, Jochen M. Kuester, Ksenia Ryndina, Jussi H. Vanhatalo, Michael S. Wahler and Olaf W. Zimmermann. *Automatic Compostion of Model Transformations (Patent Application Number 20080229276)*, 2008. 243

[Koschke 2000] R. Koschke and T. Eisenbarth. *A framework for experimental evaluation of clustering techniques*. In 8th International Workshop on Program Comprehension (IWPC'00), pages 201–210. IEEE, 2000. 180

[Kuramochi 2005] Michihiro Kuramochi and George Karypis. *Finding Frequent Patterns in a Large Sparse Graph*. Data Min. Knowl. Discov., vol. 11, no. 3, pages 243–271, 2005. 39, 40, 130, 132, 137, 245

[Kuster 2008] Jochen Kuster, Christian Gerth, Alexander Forster and Gregor Engels. *Detecting and Resolving Process Model Differences in the Absence of a Change Log*. In Business Process Management, volume 5240, pages 244–260. Springer, 2008. 27

[Lago 2009] Patricia Lago, Henry Muccini and Hans van Vliet. *A scoped approach to traceability management*. Journal of Systems and Software, vol. 82, no. 1, pages 168–182, 2009. 70

[Lam 2005] Wing Lam. *Investigating success factors in enterprise application integration: a case-driven analysis*. European Journal of Information Systems, vol. 14, no. 2, pages 175–187(13), 2005. 169, 280, 287

[Land 2003] R. Land and I. Crnkovic. *Software systems integration and architectural analysis - a case study*. In International Conference on Software Maintenance (ICSM'03, pages 338–347. IEEE, 2003. 140, 282

[Land 2007] Rikard Land and Ivica Crnkovic. *Software systems in-house integration: Architecture, process practices, and strategy selection*. Information and Software Technology, vol. 49, no. 5, pages 419–444, 2007. 12, 154, 157, 286

[Lankhorst 2005] Marc Lankhorst. Enterprise architecture at work. modelling, communication and analysis. The Enterprise Engineering Series. Springer, 2005. 70, 143, 154, 286

[Lanz 2009] Andreas Lanz, Barbara Weber and Manfred Reichert. *Time Patterns for Process-aware Information Systems: A Pattern-based Analysis - Revised version*. Technical report, University of Ulm, 2009. 54, 85

[Lawler 1980] E. L. Lawler, J. K. Lenstra and A. H. G. Rinnooy Kan. *Generating All Maximal Independent Sets: NP-Hardness and Polynomial-Time Algorithms*. SIAM Journal on Computing, vol. 9, no. 3, pages 558–565, 1980. 132

[Leicht 2006] E.A. Leicht, P. Holme and M.E. Newman. *Vertex similarity in networks*. Physical review. E, Statistical, nonlinear, and soft matter physics, vol. 73, no. 2, pages 026120–1:026120–10, 2006. 97

[Levi 2002] Keith Levi and Ali Arsanjani. *A goal-driven approach to enterprise component identification and specification*. Commun. ACM, vol. 45, no. 10, pages 45–52, 2002. 13

[Li 2003] Y. Li, Z. A. Bandar and D. McLean. *An approach for measuring semantic similarity between words using multiple information sources*. IEEE Transactions on Knowledge and Data Engineering, vol. 15, no. 4, pages 871–882, 2003. 127, 181

[Li 2006] Y. Li, D. McLean, Z. A. Bandar, J. D. O'Shea and K. Crockett. *Sentence similarity based on semantic nets and corpus statistics*. Knowledge and Data Engineering, IEEE Transactions on, vol. 18, no. 8, pages 1138–1150, 2006. 125, 126, 181, 196, 198

[Li 2008] C. Li, M. Reichert and A. Wombacher. *On Measuring Process Model Similarity Based on High-Level Change Operations*. In 27th Int. Conf. on Conceptual Modeling (ER'08), volume 5231 of *LNCS*, pages 248–264. Springer, 2008. 30

[Li 2009] Chen Li, Manfred Reichert and Andreas Wombacher. *Discovering Reference Models by Mining Process Variants Using a Heuristic Approach*. In Business Process Management, volume 5701 of *LNCS*, pages 344–362. 2009. 30, 31, 243

[Linthicum 2000] David S. Linthicum. Enterprise application integration. Addison-Wesley Professional, 2000. 12, 142, 143

[Linthicum 2004] David S. Linthicum. Next generation application integration: From simple information to web services. Addison-Wesley, 2004. 12

[Lohmann 2007] N. Lohmann, P. Massuthe and K. Wolf. *Operating Guidelines for Finite-State Services*. In Petri Nets and Other Models of Concurrency (ICATPN'07), volume 4546, pages 321–341. Springer, 2007. 241, 242

[Lu 2008] Ruopeng Lu, Shazia Sadiq and Guido Governatori. *Measurement of Compliance Distance in Business Processes*. Inf. Sys. Manag., vol. 25, no. 4, pages 344–355, 2008. 140, 159, 225, 282

[Luftman 1999] J. Luftman and T. Brier. *Achieving and sustaining business-IT alignment*. California management review, vol. 42, no. 1, pages 109–122, 1999. 143

[Malone 2003] T. W. Malone, K. Crowston and G. A. Herman. Organizing business knowledge: the mit process handbook. MIT press, 2003. 54, 237

[Massuthe 2005] P. Massuthe and K. Schmidt. *Operating guidelines - an automata-theoretic foundation for the service-oriented architecture*. In Fifth International Conference on Quality Software, 2005. (QSIC'05)., pages 452–457. IEEE, 2005. 242

[Massuthe 2007] Peter Massuthe and Wolf Karsten. *An algorithm for matching nondeterministic services with operating guidelines*. International Journal of Business Process Integration and Management, vol. 2, no. 2, pages 81–90, 2007. 28, 242

[McGuinness 2000] D. L. McGuinness, R.Fikes, J. Rice and S. Wilder. *An Environment for Merging and Testing Large Ontologies*. In 7th Int. Conf. on Principles of Knowledge Representation and Reasoning, pages 483–493, Breckenridge, CO, 2000. Morgan Kaufmann. 122

[McKay 1981] Brendan D. McKay. *Practical graph isomorphism*. Congressus Numerantium, vol. 30, pages 45–87, 1981. 34

[Medvidovic 2000] Nenad Medvidovic and Richard N. Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages*. IEEE Trans. Softw. Eng., vol. 26, no. 1, pages 70–93, 2000. 18

[Melnik 2002] S. Melnik, H. Garcia-Molina and E. Rahm. *Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching*. In 18th Int. Conf. on Data Engineering (ICDE'02), pages 117–128, San Jose, CA., 2002. IEEE. 36, 183, 212

[Mendling 2007] Jan Mendling, Gustaf Neumann and Wil van der Aalst. *On the correlation between process model metrics and errors*. In Tutorials, posters, panels and industrial contributions at the 26th int. conf. on Conceptual modeling, volume 83, pages 173–178, Auckland, NZ, 2007. Australian Computer Society, Inc. 156, 287

[Messmer 2000] Bruno T. Messmer and Horst Bunke. *Efficient Subgraph Isomorphism Detection: A Decomposition Approach*. IEEE Trans. on Knowl. and Data Eng., vol. 12, no. 2, pages 307–323, 2000. 36, 137

[Miller 1995] George A. Miller. *WordNet: a lexical database for English*. Commun. ACM, vol. 38, no. 11, pages 39–41, 1995. 211

[Mitra 2005] Tilak Mitra. *Business-driven Development*, 2005. Available from: `http://www-128.ibm.com/developerworks/webservices/library/ws-bdd/index.html`. 14

[Monroe 1997] Robert T. Monroe, Andrew Kompanek, Ralph Melton and David Garlan. *Architectural Styles, Design Patterns, and Objects*. IEEE Software, vol. 14, no. 1, pages 43–52, 1997. 13

[NACHA 2010] NACHA. *The Electronic Payments Association (NACHA)*, 2010. Available from: http://www.nacha.org. 59, 161, 169

[Navigli 2009] Roberto Navigli. *Word sense disambiguation: A survey*. ACM Comput. Surv., vol. 41, no. 2, pages 1–69, 2009. 125

[Noy 2003] Natalya F. Noy and Mark A. Musen. *The PROMPT suite - interactive tools for ontology merging and mapping*. International Journal of Human Computer Studies, vol. 59, no. 6, pages 983–1024, 2003. 122

[OASIS 2007] OASIS. *Web Services Business Process Execution Language version 2.0*, 2007. Available from: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. 14, 67, 69, 74

[OMG 2006] OMG. *UML Profile and Metamodel for Services (UPMS) RFP*, 2006. Available from: http://www.omg.org/cgi-bin/doc?soa/06-09-09. 54

[OMG 2007] OMG. *UML Version 2.1.2 (Normative). Infrastructure and Superstructure Specification. OMG*, 2007. Available from: http://www.omg.org/spec/UML/2.1.2/. 14, 52, 54, 174

[OMG 2008a] OMG. *Business Process Definition MetaModel (BPDM), Version 1.0*, 2008. Available from: http://www.omg.org/spec/BPDM/1.0/. 54

[OMG 2008b] OMG. *Business Process Modeling Notation (BPMN) version 1.1. OMG*, 2008. Available from: http://www.omg.org/spec/BPMN/1.1/. x, 14, 52, 67, 69, 74, 162, 174, 195

[OMG 2009a] OMG. *Business Process Model and Notation (BPMN). FTF Beta 1 for version 2.0*, 2009. Available from: http://www.omg.org/spec/BPMN/2.0/. ix, 76, 77

[OMG 2009b] OMG. *Service oriented architecture Modeling Language (SOAML). Specification for the UML Profile and Metamodel for Services (UPMS). Version 1.0 Beta 1.0*, 2009. Available from: http://www.omg.org/spec/SoaML/1.0/Beta1/PDF/. 53, 174

[Oracle 2008] Oracle. *State of the Business Process Management Market 2008*. Technical report, Oracle Corporation, 2008. 144, 145, 239

[Ouyang 2007] C. Ouyang, M. Dumas, A. H. M. ter Hofstede and W. M. P. van der Aalst. *Pattern-based translation of BPMN process models to BPEL web services*. International Journal of Web Services Research (JWSR), vol. 5, no. 1, pages 42–62, 2007. 14, 65, 69

[Ouyang 2009] Chun Ouyang, Marlon Dumas, M. P. Van Der Aalst Wil, Arthur H. M. Ter Hofstede and Jan Mendling. *From business process models to process-oriented software systems*. ACM Trans. Softw. Eng. Methodol., vol. 19, no. 1, pages 1–37, 2009. 169, 174

[Pahl 2006] Claus Pahl and Ronan Barrett. *Layered Patterns in Modelling and Transformation of Service-Based Software Architectures*. In Software Architecture, volume 4344 of *LNCS*, pages 144–158. Springer, 2006. 65

[Pahl 2007] Claus Pahl, Simon Giesecke and Wilhelm Hasselbring. *An Ontology-based Approach for Modelling Architectural Styles*. In First European Conference on Software Architecture (ECSA'07), volume 4758 of *LNCS*, pages 60–75. Springer, 2007. 18, 56

[Pahl 2009a] Claus Pahl, Simon Giesecke and Wilhelm Hasselbring. *Ontology-based modelling of architectural styles*. Information and Software Technology, vol. 51, no. 12, pages 1739–1749, 2009. 63, 243

[Pahl 2009b] Claus Pahl, Yaoling Zhu and Veronica Gacitua-Decar. *A Template-driven Approach for Maintainable Service-oriented Information Systems Integration*. International Journal of Software Engineering and Knowledge Engineering, vol. 19, no. 7, pages 889–912, 2009. 65

[Palmer 2009] N. Palmer. *2009 BPM - State of the Markey Report*. Technical report, Transf. & Innov., BPM Educ. Assoc., BPM.com and the Workflow Management Coalition, 2009. 239

[Papazoglou 2006a] M. P. Papazoglou and W. J. van den Heuvel. *Service-Oriented Design and Development Methodology*. Int. J. of Web Engineering and Technology (IJWET), vol. 2, pages 412 – 442, 2006. 13, 20, 41

[Papazoglou 2006b] Mike Papazoglou and Benedikt Kratz. *A Business-Aware Web Services Transaction Model*. In Service-Oriented Computing (ICSOC'06), volume 4294, pages 352–364. Springer, 2006. 20, 22, 41

[Papazoglou 2007] M. Papazoglou and W.-J. var den Heuvel. *Service oriented architectures: approaches, technologies and research issues*. IJVLDB, vol. 16, no. 3, pages 389–415, 2007. 284

[Pedersen 2008] Ted Pedersen, Siddharth Patwardhan, Satanjeev Banerjee and Jason Michelizzi. *WordNet::Similarity*, 2008. Available from: `http://search.cpan.org/dist/WordNet-Similarity/doc/intro.pod`. 211

[Pinzger 2005] Martin Pinzger, Harald Gall and Michael Fischer. *Towards an Integrated View on Architecture and its Evolution*. ENTCS, vol. 127, no. 3, pages 183–196, 2005. 120

[ProM 2009] ProM. *Process Mining - Research Tools Application*, 2009. Available from: `http://www.processmining.org`. 243, 245

[Puschmann 2004] Thomas Puschmann and Rainer Alt. *Enterprise application integration systems and architecture - the case of the Robert Bosch Group*. Journal of Enterprise Information Management, vol. 17, no. 2, pages 105 – 116, 2004. 169, 280, 287

[Rabhi 2007] Fethi Rabhi, Hairong Yu, Feras Dabous and Sunny Wu. *A service-oriented architecture for financial business processes*. Information Systems and E-Business Management, vol. 5, no. 2, pages 185–200, 2007. 132

[Ralyte 2008] Jolita Ralyte, Manfred A. Jeusfeld, Per Backlund, Harald Kuhn and Nicolas Arni-Bloch. *A knowledge-based approach to manage information systems interoperability*. Information Systems, vol. 33, no. 7-8, pages 754–784, 2008. 16, 18

[Rapanotti 2004] L. Rapanotti, J. G. Hall, M. Jackson and B. Nuseibeh. *Architecture-driven problem decomposition*. In 12th Int. Requirements Eng. Conf., pages 80–89. IEEE, 2004. 16, 18

[Rech 2009] Jorg Rech and Christian Bunse. Model-driven software development: Integrating quality assurance. IGI Global, 2009. 282

[Reichert 1998] Manfred Reichert and Peter Dadam. *Adeptflex - Supporting Dynamic Changes of Workflows Without Losing Control*. Journal of Intelligent Information Systems, vol. 10, no. 2, pages 93–129, 1998. 30

[Riehle 1996] Dirk Riehle and Heinz Zullighoven. *Understanding and using patterns in software development*. Theor. Pract. Object Syst., vol. 2, no. 1, pages 3–13, 1996. 16

[Rising 2007] Linda Rising. *Understanding the Power of Abstraction in Patterns*. IEEE Softw., vol. 24, no. 4, pages 46–51, 2007. 57

[Ross 2008a] A. M. Ross and D. H. Rhodes. *Architecting Systems for Value Robustness: Research Motivations and Progress*. In IEEE Systems Conference, pages 1–8. IEEE, 2008. 284

[Ross 2008b] A. M. Ross, D. H. Rhodes and D. E. Hastings. *Defining changeability: Reconciling flexibility, adaptability, scalability, modifiability, and robustness for maintaining system lifecycle value*. Systems Engineering, vol. 11, no. 3, pages 246–262, 2008. 283

[Ross 2009] A. M. Ross, D. H. Rhodes and D. E. Hastings. *Using Pareto Trace to determine system passive value robustness*. In IEEE Systems Conference, pages 285–290. IEEE, 2009. 171, 174, 285

[Rozenberg 1997] G Rozenberg, editor. Handbook of graph grammars and computing by graph transformation, volume 1: Foundations. World Scientific Publishing Co., 1997. 37

[Sadiq 2000] Wasim Sadiq and Maria E. Orlowska. *Analyzing process models using graph reduction techniques*. Information Systems, vol. 25, no. 2, pages 117–134, 2000. 74, 269

[Sartipi 2001] K. Sartipi and K. Kontogiannis. *A graph pattern matching approach to software architecture recovery*. In K. Kontogiannis, editor, 17th IEEE International Conference on Software Maintenance (ICSM'01), pages 408–419. IEEE, 2001. 23, 180

[Schaeffer 2007] Satu Elisa Schaeffer. *Graph clustering*. Computer Science Review, vol. 1, no. 1, pages 27–64, 2007. 245

[SEI 2010] SEI. *Software Engineering Institute - Carnegie Mellon University*, 2010. Available from: `http://www.sei.cmu.edu/`. 5

[Shaw 1996a] M. Shaw and D. Garlan. Software architecture. Prentice Hall, Upper Saddle River, NJ, 1996. 144, 145

[Shaw 1996b] Mary Shaw. *Some patterns for software architectures*. In Pattern languages of program design. Volume 2, pages 255–269. Addison-Wesley Longman Publishing Co., Inc., 1996. 18

[Shaw 2002] M. Shaw. *What Makes Good Research in Software Engineering?'*. International Journal of Software Tools for Technology Transfer, vol. 4, no. 1, pages 1–7, 2002. 3

[Smirnov 2009] Sergey Smirnov, Matthias Weidlich, Jan Mendling and Mathias Weske. *Action Patterns in Business Process Models*. In Service-Oriented Computing, volume 5900, pages 115–129. Springer, 2009. 16, 19, 20, 41, 54, 98, 237, 239

[Spanoudakis 2010] George Spanoudakis and Andrea Zisman. *Discovering Services during Service-Based System Design Using UML*. IEEE Transactions on Software Engineering, vol. 36, no. 3, pages 371–389, 2010. 15, 28, 31

[SparxSystems 2010] SparxSystems. *Enterprise Architect*, 2010. Available from: `http://www.sparxsystems.com/products/ea/`. 174, 175, 236

[Steen 2005] M. W. A. Steen, P. Strating, M. M. Lankhorst, H. ter Doest and M. E. Iacob. *Service-Oriented Enterprise Architecture*. In Z. Stojanovic and A. Dahanayake, editors, Service-Oriented Software System Engineering: Challenges and Practices, pages 132–154. Idea Group, 2005. 24

[Taentzer 2004] Gabriele Taentzer. *AGG: A Graph Transformation Environment for Modeling and Validation of Software*. In Applications of Graph Transformations with Industrial Relevance, volume 3062, pages 446–453. Springer, 2004. 38

[Taentzer 2005] Gabriele Taentzer and Arend Rensink. *Ensuring Structural Constraints in Graph-Based Models with Type Inheritance*. In Fundamental Approaches to Software Engineering, volume 3442 of *LNCS*, pages 64–79. Springer, 2005. 76, 89, 269, 272

[Tandon 2007] R. Tandon. *The Role of Solution Architects in Systems Integration*. IT Professional, vol. 9, no. 2, pages 26–33, 2007. 141

[Taylor 1996] R. N. Taylor, N. Medvidovic, K. M. Anderson, Jr. Whitehead E. J., J. E. Robbins, K. A. Nies, P. Oreizy and D. L. Dubrow. *A component- and message-based architectural style for GUI software*. IEEE TSE, vol. 22, no. 6, pages 390–406, 1996. 18

[Themistocleous 2004] M. Themistocleous. *Justifying the decisions for EAI implementations: a validated proposition of influential factors*. Journal of Enterprise Information Management, vol. 17, no. 2, pages 85–104, 2004. 12, 283

[Thom 2007] L. H. Thom, C. Iochpe and M. Reichert. *Workflow Patterns for Business Process Modeling*. In B. Pernici and J.A. Gulla, editors, 8th Int'l Workshop on Business Process Modeling, Development, and Support (BPMDS'07), in: CAISE'06 Workshops, volume 1, Trondheim, Norway., 2007. Tapir Academic Press. 16, 19, 20

[Thom 2009] L. H. Thom, M. Reichert and C. Iochpe. *Activity patterns in process-aware information systems: basic concepts and empirical evidence*. Int. J. of Business Process Integration and Management, vol. 4, no. 2, pages 93–110, 2009. 20, 41, 54, 98, 237, 239

[Tran 2006] Hanh Tran, Bernard Coulette and Bich Dong. *A UML-Based Process Meta-model Integrating a Rigorous Process Patterns Definition*. In Product-Focused Software Process Improvement, volume 4034 of *LNCS*, pages 429–434. Springer, 2006. 16, 19

[Tran 2007] Hanh Nhi Tran, Bernard Coulette and Dong Thi Bich Thuy. *Broadening the Use of Process Patterns for Modeling Processes*. In 19th Int. Conf. on Software Engineering & Knowledge Engineering (SEKE'2007), pages 57–62, Boston, MA, 2007. Knowledge Systems Inst. Grad. School. 16, 19

[Tsantalis 2006] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides and S. T. Halkidis. *Design Pattern Detection Using Similarity Scoring*. IEEE Transactions on Software Engineering, vol. 32, no. 11, pages 896–909, 2006. ix, 23, 42, 43, 44

[Ullmann 1976] J. R. Ullmann. *An Algorithm for Subgraph Isomorphism*. J. ACM, vol. 23, no. 1, pages 31–42, 1976. 35

[Umar 2009] Amjad Umar and Adalberto Zordan. *Reengineering for service oriented architectures: A strategic decision model for integration versus migration*. Journal of Systems and Software, vol. 82, no. 3, pages 448–462, 2009. 12

[Valiente 1997] Gabriel Valiente and Conrado Martinez. *An Algorithm for Graph Pattern-Matching*. In 4th South American Workshop on String Processing. Int. Informatics Series, volume 8, pages 180–197, 1997. 36, 237

[Vanderfeesten 2007] I. Vanderfeesten, J. Cardoso, J. Mendling, H. Reijers and W. van der Aalst. *Quality Metrics for Business Process Models*. In L. Fischer, editor, 2007 BPM I& Workflow Handbook, pages 179–190. WfMC, Lighthouse Point, FL, 2007. 287

[Vanetik 2002] N. Vanetik, E. Gudes and S. E. Shimony. *Computing frequent graph patterns from semistructured data*. In IEEE Int. Conf. on Data Mining (ICDM'02), pages 458–465, 2002. 131

[Vanhooff 2007] B. Vanhooff, S. Van Baelen, W. Joosen and Y. Berbers. *Traceability as Input for Model Transformations*. In 3rd ECMDA Traceability Workshop, pages 37–46, Haifa, Israel, 2007. SINTEF. 69

[Varro 2002] Daniel Varro, Gergely Varro and Andras Pataricza. *Designing the automatic transformation of visual languages*. Science of Computer Programming, vol. 44, no. 2, pages 205–227, 2002. 38

[Varro 2005] Gergely Varro, Andy Schurr and Daniel Varro. *Benchmarking for Graph Transformation*. In IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC'05), pages 79–88, Washington, USA, 2005. IEEE. 38, 243

[Varro 2006a] Gergely Varro, Katalin Friedl and Daniel Varro. *Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans*. ENTCS, vol. 152, pages 191–205, 2006. 38

[Varro 2006b] Gergely Varro, Katalin Friedl and Daniel Varro. *Implementing a Graph Transformation Engine in Relational Databases*. Software and Systems Modeling, vol. 5, no. 3, pages 313–341, 2006. 38

[Vokac 2004] M. Vokac, W. Tichy, D. I. K. Sjoberg, E. Arisholm and M. Aldrin. *A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns - A Replication in a Real Programming Environment*. Empirical Software Engineering, vol. 9, no. 3, pages 149–195, 2004. 144

[Wang 1995] J. T. L. Wang, K. Zhang and D. Shasha. *Pattern matching and pattern discovery in scientific, program, and document databases*. In M. Carey and D. Schneider, editors, ACM SIGMOD Int. Conf. on Management of Data, volume 24, page 487, San Jose, CA, 1995. ACM. 39, 137

[Weber 2007] Barbara Weber, Stefanie Rinderle and Manfred Reichert. *Change Patterns and Change Support Features in Process-Aware Information Systems*. In Advanced Information Systems Engineering, pages 574–588. 2007. 30

[Weber 2008] Barbara Weber, Manfred Reichert and Stefanie Rinderle-Ma. *Change patterns and change support features - Enhancing flexibility in process-aware information systems*. Data & Knowledge Engineering, vol. 66, no. 3, pages 438–466, 2008. 54

[Weyuker 1988] E. J. Weyuker. *Evaluating software complexity measures*. IEEE Transactions on Software Engineering, vol. 14, no. 9, pages 1357–1365, 1988. 287

[White 2005] Stephen White. *Using BPMN to Model a BPEL Process*, 2005. Available from: `http://www.bpmn.org/Documents/MappingBPMNto20BPELExample.pdf`. 14

[Wierzbicki 1982] Andrzej P. Wierzbicki. *A mathematical basis for satisficing decision making*. Mathematical Modelling, vol. 3, no. 5, pages 391–405, 1982. 283

[Wombacher 2004] A. Wombacher, P. Fankhauser, B. Mahleko and E. Neuhold. *Matchmaking for business processes based on choreographies*. In IEEE International Conference on e-Technology, e-Commerce and e-Service - EEE '04, pages 359–368, 2004. 27, 28, 31

[Wombacher 2006] A. Wombacher and M. Rozie. *Evaluation of Workflow Similarity Measures in Service Discovery*. In M. Schoop, C. Huemer, M. Rebstock and M. Bichler, editors, Service Oriented Electronic Commerce, volume 80, pages 51–71. GI, 2006. ix, 42, 43, 44

[Woodley 2005] T. Woodley and S. Gagnon. *BPM and SOA: Synergies and Challenges*. In Web Information Systems Engineering (WISE'05), volume 3806 of *LNCS*, pages 679–688. Springer, 2005. 12, 33

[WordNet 2010] WordNet. *WordNet - A Lexical Database for the English Language*, 2010. Available from: `http://wordnet.princeton.edu/`. 125

[WP 1999] WP. *Workflow Patterns*, 1999. Available from: `www.workflowpatterns.com/`. 54

[Zdun 2006] U. Zdun, C. Hentrich and W. M. P. van der Aalst. *A survey of patterns for Service-Oriented Architectures*. International Journal of Internet Protocol Technology, vol. 1, no. 3, pages 132–143, 2006. 65, 171, 174

[Zdun 2007a] U. Zdun. *Systematic pattern selection using pattern language grammars and design space analysis*. Software Practice and Experience, vol. 37, no. 9, pages 983–1016, 2007. 21

[Zdun 2007b] Uwe Zdun and Schahram Dustdar. *Model-driven and pattern-based integration of process-driven SOA models*. International Journal of Business Process Integration and Management, vol. 2, no. 2, pages 109 – 119, 2007. 143

[Zhang 2005] Zhuopeng Zhang, Ruimin Liu and Hongji Yang. *Service Identification and Packaging in Service Oriented Reengineering*. In 7th International Conference on Software Engineering and Knowledge Engineering (SEKE'07), pages 241–249, 2005. 26

[Zhang 2008] L. J. Zhang, N. Zhou, Y. M. Chee, A. Jalaldeen, K. Ponnalagu, R. R. Sindhgatta, A. Arsanjani and F. Bernardini. *SOMA-ME: a platform for the model-driven design of SOA solutions*. IBM Syst. J., vol. 47, no. 3, pages 397–413, 2008. 14, 25, 237

[Zhang 2009] Liang-Jie Zhang, Zhi-Hong Mao and Nianjun Zhou. *Design Quality Analytics of Traceability Enablement in Service-Oriented Solution Design Environment*. In IEEE International Conference on Web Services (ICWS'09)., pages 944–951, Los Angeles, CA, 2009. IEEE. 25, 237

[Zhao 2007] Chunying Zhao, Jun Kong, Jing Dong and Kang Zhang. *Pattern-based design evolution using graph transformation*. Journal of Visual Languages and Computing, vol. 18, no. 4, pages 378–398, 2007. 22

[Zimmermann 2007] Olaf Zimmermann, Jonas Grundler, Stefan Tai and Frank Leymann. *Architectural Decisions and Patterns for Transactional Workflows in SOA*. In Service-Oriented Computing (ICSOC'07), pages 81–93. 2007. 21, 22

[Zundorf 1996] A. Zundorf. *Graph Pattern Matching in PROGRES*. In H. Ehrig, G. Engels and G. Rozenberg, editors, Selec. papers from the 5th Int. Workshop on Graph Gramars and their Application to Comp. Sci., volume 1073, pages 454–468. Springer, 1996. 37, 38

Appendix A

Background on Graphs

Contents

Graphs emerge as a natural representation for process models [Ehrig 1999a],[Sadiq 2000]. Graphs can capture both structure and behaviour, and allow abstractions such as patterns to be related to process models. This appendix provides an introduction to basic concepts of graphs. In particular, the notions of typed attributed graph and graph homomorphism are used to formally describe pattern instances and their relation to patterns and models. Graphs and graph homomorphisms are introduced using the definitions from [Hell 2004] and [Fiala 2007], and for attributed typed graphs, definitions from [Corradini 1996], [Heckel 2002], [Ehrig 2004], [Taentzer 2005] and [Ehrig 2006b] are utilised.

A.1 Digraphs and Undirected Graphs

A *digraph* G is a finite set $V = V(G)$ of vertices, together with a binary relation $E = E(G)$ on V. The elements (u, v) of E are called the *arcs* of G. A digraph is symmetric, or reflexive, or irreflexive, if the relation E is symmetric, or reflexive, or irreflexive, respectively. Symmetric digraphs are also referred as *undirected* graphs. Formally, a graph G is a set $V = V(G)$ of vertices together with a set $E = E(G)$ of edges, each of which is a two-element set of vertices. Edges that only consist of one vertex are called *loops*. If every vertex in a graph has a loop, the graph is a reflexive graph. Suppose G is a graph with loops allowed, the corresponding symmetric digraph of G is obtained from G by replacing each edge $\{u, v\}$ with the two arcs (u, v), (v, u), and each loop $\{w\}$ with the arc (w, w). Figure A.1 from [Hell 2004] illustrates the relations between graph classes through examples.

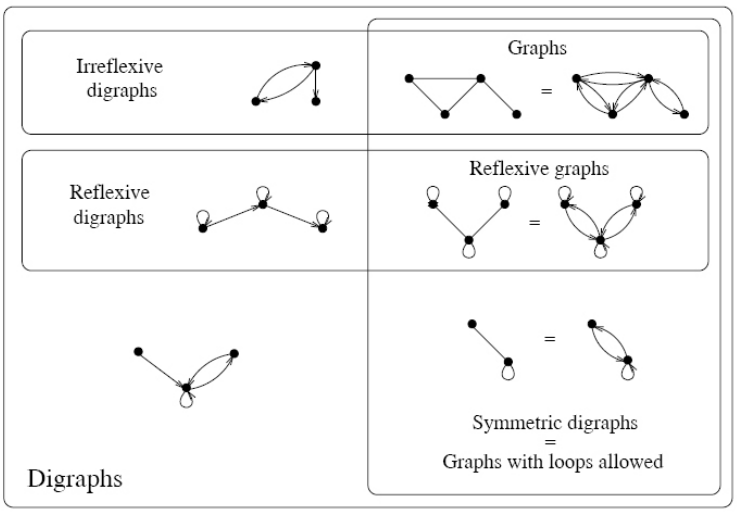


Figure A.1: Digraphs and classes of graphs (from [Hell 2004]).

Walks, Cycles and Paths [Hell 2004]. A walk in a graph G is a sequence of vertices $v_0, v_1, \cdots, v_k$ of G such that v_{i-1} and v_i are adjacent, for each $i = 1, 2, \cdots, k$. A walk is closed if $v0 = vk$. A path in G is a walk in which all the vertices are distinct. The integer k is called the length of the walk, respectively the length of the path. The graph with vertices $0, 1, \cdots, k$ and edges $\{0,1\}, \{1,2\}, \cdots, \{(k-1), k\}$ is called the path P_k. Note that P_k has $k+1$ vertices and k edges. A *cycle* in a graph G is a sequence of distinct vertices $v_1, v_2, \cdots, v_k$ of G such that each v_i, $i = 2, 3, \cdots, k$, is adjacent to v_{i-1}, and v_1 is adjacent to v_k. A graph with vertices $0, 1, \cdots, k-1$ and edges $\{i, (i+1)\}$ for $i = 0, 1, \cdots, k-1$ (with addition modulo k) is called the cycle C_k. C_k has k vertices and k edges. A cycle is a closed walk, and thus the definition of length is still applicable. Directed paths and cycles $\overrightarrow{P_k}$ and $\overrightarrow{C_k}$ are defined exactly as the graphs P_k and C_k, only this time $i(i+1)$ are arcs and not edges.

A graph G is called *connected* if for every pair of distinct vertices u and v, there exists a *path* connecting u and v.

Subgraph and Induced Subgraph. A graph G is a *subgraph* of H, and H is a *supergraph* of G, if $V_G \subseteq V_H$ and $E_G \subseteq E_H$. A graph G is called an *induced subgraph* of H if G is a subgraph of H and it contains all the arcs (edges) of H amongst the vertices in G. A *clique* in a graph G is a complete subgraph of G, i.e., for a clique in the undirected graph $G = (V, E)$, the set of vertices in the clique $C \subset V$ is such that for every two vertices in C there exists an edge connecting the two vertices in E.

Neighbourhood. For a vertex u in a graph G, the set of all vertices adjacent to u are called the *neighbourhood* of u denoted by $N_G(u)$ and thus $N_G(u) = \{v | \{u, v\} \in E_G\}$.

A.2 Graph Homomorphisms

Let G and H be any digraphs. A homomorphism of G to H, written as $f : G \to H$, is a mapping $f : V(G) \to V(H)$ such that $f(u)f(v) \in E(H)$ whenever $uv \in E(G)$. Note that uv and $f(u)f(v)$ denote an arc (or edge) in $E(G)$ and $E(H)$, respectively. The composition $f \circ g$ of homomorphisms $g : G \to H$ and $f : H \to K$ is a homomorphism of G to K. Thus, the binary relation *is homomorphic to* on the set of digraphs is transitive. Note that homomorphisms of graphs preserve adjacency, while homomorphisms of digraphs also preserve the directions of the arcs. Therefore, a homomorphism of digraphs $G \to H$ is also a homomorphism of the underlying graphs, but not conversely.

A homomorphism $f : G \to H$ is a mapping of $V(G)$ to $V(H)$, but since it preserves adjacency it also naturally defines a mapping f' of $E(G)$ to $E(H)$ by setting $f'(uv) = f(u)f(v)$ for all $uv \in E(G)$. A homomorphism $f : G \to H$ is called vertex-injective, or vertex-surjective, or vertex-bijective, if the mapping $f : V(G) \to V(H)$ is injective, or surjective, or bijective respectively. The same homomorphism $f : G \to H$ is called edge-injective, or edge-surjective, or edge-bijective, if the mapping $f' : E(G) \to E(H)$ is injective, or surjective, or bijective, respectively. Thus, a homomorphism f is an injective homomorphism, or a surjective homomorphism or a bijective homomorphism, if it is both vertex- and edge-injective, or surjective, or bijective, respectively.

Since graph homomorphisms are edge preserving vertex mapping between two graphs, the mapping have the following property: if two vertices form an edge in the source graph then their images form an edge in the target graph (which might be the same graph) [Fiala 2005]. Thus, $f : V_G \to V_H$ is a graph homomorphism if and only if $(f(u), f(v)) \in E_H$ for all pairs $(u, v) \in E_G$.

Locally Constrained Graph Homomorphism (LCGH). LCGH is a special kind of graph homomorphism where the image of the neighbourhood of a vertex in a source graph is contained in the neighbourhood of the image of the vertex in the target graph [Fiala 2008], i.e. $f(N_G(u)) \subseteq N_H(f(u))$ holds for every vertex $u \in V_G$ whenever $f : V_G \to V_H$ is a homomorphism from G to H.

If the neighbourhood of any vertex of the source graph is mapped bijectively, injectively or surjectively to the neighbourhood of its image in the target graph, the homomorphism is called locally bijective, injective or surjective, respectively [Fiala 2008]. In particular,

- $G \xrightarrow{B} H$ if there exist a *locally bijective homomorphism* $f : V_G \to V_H$ that satisfies for all $u \in V_G : f(N_G(u)) = N_H(f(u))$ and $|f(N_G(u))| = |N_G(u)|$.
- $G \xrightarrow{I} H$ if there exist a *locally injective homomorphism* $f : V_G \to V_H$ that satisfies for

all $u \in V_G : |f(N_G(u))| = |N_G(u)|$.

- $G \xrightarrow{S} H$ if there exist a *locally surjective homomorphism* $f : V_G \to V_H$ that satisfies for all $u \in V_G : f(N_G(u)) = N_H(f(u))$.

Note that for the mappings above, locally bijective homomorphism is both locally injective and surjective. The mappings are also known in the literature as (full) covering projections (bijective), or as partial covering projections (injective), or as role assignments (surjective). Additionally, any locally surjective homomorphism f from a graph G to a connected graph H is globally surjective, and any locally injective homomorphism f from a connected graph G to a forest H is globally injective [Fiala 2007].

Observation: Note that in concrete categories considered in universal algebra such as groups, rings, modules, etc., morphisms are called homomorphisms.

Homomorphisms for Walks, Cycles and Paths [Hell 2004]. A mapping $f : V(P_k) \to V(G)$ is a homomorphism of P_k to G if and only if the sequence $f(0), f(1), \cdots, f(k)$ is a walk in G. Homomorphisms of G to H map paths in G to walks in H, and hence do not increase distances. Thus, for $d_G(u,v)$ denoting the distance (length of a shortest path) from u to v in G and $f : G \to H$ an homomorphism, $d_H(f(u), f(v)) \leq d_G(u,v)$, for any two vertices u, v of G. A mapping $f : V(C_k) \to V(G)$ is a homomorphism of C_k to G if and only if $f(0), f(1), \cdots, f(k-1)$ is a closed walk in G.

A.3 Typed Graphs and Morphisms

A *typed graph* G is defined by the graph G and a graph morphism between G and a *graph type* TG. TG not only defines the types of G but also the relations between types; and the morphism $G \to TG$ provides additional information than a simple coloring function providing colors (or types) to G, since the structure of TG has to be maintained in G [Corradini 1996].

The category having graphs as objects and graph morphisms as arrows is called $GRAPH$ and the category of graphs typed over a type graph TG, written as $TG - GRAPH$, has typed graphs as objects and typed graph morphisms as arrows. Typed graphs are pairs $\langle G, m \rangle$ where G is a graph and $m : G \to GT$ is a graph morphism. Typed graph morphisms (arrows) $f : \langle G, m \rangle \to \langle G', m' \rangle$ are graph morphisms $f : G \to G'$ such that $m' \circ f = m$ [Corradini 1996].

Type Graph with Inheritance and Multiplicities [Taentzer 2005].
Type Graph with Inheritance. A type graph with inheritance is a triple $TGI = (TG, I, A)$ consisting of a type graph $TG = (V(TG), E(TG), src, tar)$

(with a set $V(TG)$ of vertices, a set $V(E)$ of edges, source and target functions $src, tar : E(TG) \to V(TG)$), an acyclic inheritance relation $I \subseteq V(TG) \times V(TG)$, and a set $A \subseteq V(TG)$, called abstract nodes. For each $v_1 \in V(TG)$, the inheritance *clan* is defined by $clan_I(v_1) = \{v_2 \in V(TG) | (v_2, v_1) \in I_*\}$, where I_* is the reflexive-transitive closure of I.

Flattening or Closure of Type Graphs with Inheritance to Ordinary Type Graphs. Let $TGI = (TG, I, A)$ be a type graph with inheritance, and let $TG = (V(TG), E(TG), src, tar)$, The abstract closure of TGI is the graph $\overline{TGI} = V(TG), \overline{E(TG)}, \overline{src}, \overline{tar})$ with $\overline{E(TG)} = \{(v_1, e, v_2) | e \in E(TG), v_1 \in clan_I(src(e)), v_2 \in clan_I(tar(e))\}$; $\overline{src}((v_1, e, v_2)) = v_1$; $\overline{tar}((v_1, e, v_2)) = v_2$. The closure of TGI is the graph $\widehat{TGI} = \overline{TGI}|_{V(TG)-A}$, where $\overline{TGI}|_{V(TG)-A}$ denotes the subgraph $TGI_{sub}(V), TGI_{sub}(E) = \{e \in E(TG) | src(e), tar(e) \in TGI_{sub}\}, src|_{TGI_{sub}(E)}, tar|_{TGI_{sub}(E)}$ and $TGI_{sub}(V) = V(TG) - A$.

An abstract instance graph of a type graph with inheritance TGI is an instance graph of $\overline{TGI}$. Instance graphs can be typed over the type graph with inheritance by a pair of functions, from nodes to node types and from edges to edge types, respectively. The pair of functions is called clan morphism and uniquely characterizes the type morphism into the flattened type graph.

Clan Morphism. Let $TGI = (TG, I, A)$ be a type graph with inheritance. A clan morphism from G to TGI is a pair $ctp = (ctp_{V(G)} : V(G) \to V(TG), ctp_{E(G)} : E(G) \to E(TG))$ such that for all $e \in E(G)$, $ctp_{V(G)} \circ src_G(e) \in clan_I(src_{TG} \circ ctp_{E(G)}(e))$ and $ctp_{V(G)} \circ tar_G(e) \in clan_I(tar_{TG} \circ ctp_{E(G)}(e))$. (G, ctp) is called *clan typed graph* and ctp is called concrete if $ctp^{-1}_{V(G)}(A) = \emptyset$.

Type Graph with Edge Inheritance. A type graph with edge inheritance is a tuple (TG, I, A) where $I \subseteq (TG(V) \times TG(V)) \cup (TG(E) \times TG(E))$ is an acyclic relation such that $TGI = (TG, I|_{TG(V)}, A)$ is a type graph with (vertex) inheritance, and $(e, f) \in I \cap (E(TG) \times E(TG))$ implies $src(e) \in clan_I(src(f))$ and $tar(e) \in clan_I(tar(f))$. Thus, if a type edge e inherits from another type edge f, then f can occur as an edge type only for concrete graph edges whose source and target vertex types are not in the clan of the source type.

Type Graph with Multiplicities. Type graph with multiplicities allows to restrict the class of correctly typed instance graphs to those satisfying additional constraints concerning the number of nodes and edges for each type. These constraints are expressed using *multiplicities*. *Multiplicity* is a pair $[i, j] \in \mathcal{N} \times (\mathcal{N} \cup \{*\})$ with $i \leq j$ or $j = *$ where $*$ indicates the number of vertices or edges is not constrained. The

set of multiplicities is denoted $Mult$. For an arbitrary finite set X and $[i,j] \in Mult$, $|X| \in [i,j]$ if $i \leq |X|$ and either $j = *$ or $|X| \leq j$. A type graph with multiplicities is a tuple $TGM = (TGI, m_{TGI(V)}, m_{src}, m_{tar})$ consisting of a type graph with inheritance TGI and additional functions $m_{TGI(V)} : TGI(V) \to Mult$, called node multiplicity function, and $m_{src}, m_{tar} : TGI(E) \to Mult$, called edge multiplicity functions. The satisfaction of multiplicity constraints is expressed by counting inverse images with respect to the clan typing.

A.4 Attributed Graphs and Morphisms

The following definitions and notation are extracted from [Heckel 2002].

An *attributed graph* over Σ is a pair $AG = \langle G, A \rangle$ of a graph G and a Σ-algebra A such that $|A| \subseteq V(G)$, where $|A|$ is the disjoint union of the carrier sets A_S of A, for all $s \in S$, and such that $\forall e \in E(G) : src(e) \notin |A|$. Note that a many-sorted signature $\Sigma = \langle S, OP \rangle$ consists of a set of sort symbols $s \in S$ and a family of sets of operation symbols $op : s1 \dots sn \to s \in OP$ indexed by their arities; and G is a graph with $src^G : E(G) \to V(G)$ and $tar^G : E(G) \to V(G)$, where src^G and tar^G are functions associating each arc to its source and target vertices. Let refer to an attributed graph, associated attributes and algebra as $Graph(AG) = G \backslash (|A| + Attr(AG))$, $Attr(AG) = \{e \in E(G) | tar(e) \in |A| \}$ and $Alg(AG) = A$. For attributes, data values are represented as vertices of graphs, called *data vertices* $d \in |A|$ to distinguish them from *graph vertices* $v \in V(G) \backslash |A|$. Graph vertices are linked to data vertices by edges[1] called *attributes* $a \in E(G)$ with $src(a) = v$ and $tar(a) = d$. Edges between graph vertices are called *links*.

An attributed graph morphism $f : \langle G_1, A \rangle \to \langle G_2, A \rangle$ is a pair of a σ-homomorphism $f_A = (f_S)_{s \in S} : A_1 \to A_2$ and a graph homomorphism $f_G : G_1 \to G_2$ such that $|f_A| \subseteq \bigcup_{s \in S} f_S$.

Attributed graphs and graph morphisms form a category of Σ-attributed graphs. Often, the data algebra A is fixed in advance, in that case, graphs and graph morphisms are said attributed over A.

A.5 Attributed Typed Graph

The following definitions and notation are extracted from [Heckel 2002].

An *attributed type graph* over Σ is an attributed graph $ATG = \langle TG, Z \rangle$ over Σ where Z is the final Σ-algebra Z having $Z_S = \{s\}$ for all $s \in S$. Elements of Z

[1] Note that throughout this thesis *edge* is used as a generic term to refer both, undirected edges and directed edges (arcs), and at the minimum a distinction is required.

represent the sorts of the signature which are included in TG as types for data vertices. In general, vertices and edges of TG represent vertex and arc types, while attributes in ATG are actual attribute declarations. Given an attribute declaration $a \in ATG$ and an graph vertex $v \in AG$ such that $ag(v) = src(a)$, $a(v)$ would denote the set of a-values of vs $\{d \in |A| \text{ such that } \exists e \in E(AG), src(e) = v \wedge tar(e) = d\}$.

An attribute instance graph $\langle AG, ag \rangle$ over ATG is an attributed graph AG, over the same signature, together with an attributed graph morphism $ag : AG \to ATG$. Instance graphs are usually infinite because the set of instances of the data type is infinite (e.g. the natural numbers) and each instance would be a separate vertex.

A morphism of typed attributed graphs $h : \langle AG_1, ag_1 \rangle \to \langle AG_2, ag_2 \rangle$ is a morphism of attributed graphs which preserves the typing, that is $ag_2 \circ h = ag_1$.

Attributed Typed Graph with Data Type Constraints [Ehrig 2006b]. Let $\langle ATG, Constr \rangle$ be an attributed typed graph with constraints, where $Constr$ is a suitable set of constraints and ATG an attributed typed graph. The authors of [Ehrig 2006b] consider three types of constraints: graph, data type, and graph-OCL constraints for expressing metamodels in visual languages, where the kind of constraints to consider depends on the specific visual language. Data type constraints can be given by equations or first order formulas over the data type signature Σ of ATG. An attributed graph $AG = \langle G, Z \rangle$ satisfies a data type constraint *data-constr*, if the Σ-algebra Z satisfies *data-constr*.

A.6 Graph Transformations

The following definitions and notation are extracted from [Ehrig 2006a].

Graph transformations can capture the modifications of graph-based models. A graph transformation step of a graph-based model G into a graph-based model H can be captured through a *production* rule applied to G to obtain H. There are two well-known approaches, DPO and SPO, to model rule-based graph transformations. Before relating graph transformations to changes in pattern instances, a brief introduction of the DPO and SPO approaches is provided. Figure A.2 illustrates DPO and SPO.

Graph Transformations: the Double- and Single-Pushout Approaches. In the DPO approach, a production $p = (L \leftarrow K \rightarrow R)$ is given by $p = (L, K, R)$, where L and R are the left- and right-hand side graphs and K is the intersection of L and R. The left-hand side L represents the preconditions of the rule and the right-hand side R, the postconditions. K describes the elements of the graph which has to exist

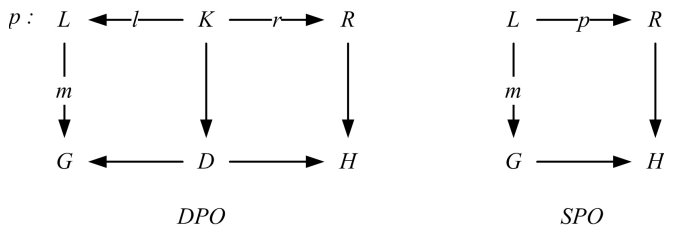


Figure A.2: Double- and single- pushout approaches to graph transformations.

to apply the rule. These elements are not changed. $L \setminus K$ describes the graph elements which are deleted and $R \setminus K$ describes the graph elements which are created. In order to derive a transformation on a host graph G, first a match m between the left-hand side L of the production p in the host graph G must to be found. The mach m is must be structure-preserving. When a direct graph transformation with a production p and a match m is performed, all the vertices and edges which are matched by $L \setminus K$ are removed from G. The the condition for m is that D should not have dangling edges, therefore the remaining structure $D := (G \setminus m(L)) \cup m(K)$ should be a graph. Thus, m has to satisfy a gluing condition such that the gluing of $L \setminus K$ and D is equal to G. The next step takes the graph D and glue it together with $R \setminus K$ to obtain the derived graph H. The morphisms from G to D and from D to H derive the transformation step from G to H.

In the SPO approach, the production $p = (L \leftarrow K \rightarrow R)$ can be considered as a partial graph morphism $p : L \rightarrow R$ with domain $dom(p) = K$. The horizontal morphisms from L to R and from G to H are partial and the vertical ones (from L to G and from R to H) are total. SPO and DPO differ in the deletion of elements in D during the transformation step. If the match $m : L \rightarrow G$ does not satisfy the gluing condition with respect to a production $p : L \leftarrow K \rightarrow R$, then p is not applicable inn DPO, but in SPO. SPO allows dangling edges to occur after the deletion of $L \setminus K$ from G but dangling edges in G are deleted, resulting a well-defined graph H.

A *graph grammar* $\mathcal{G}$ is a pair $\mathcal{G} = \langle (p : r)_{p \in Pr}, G_0 \rangle$ where $(p : r)_{p \in Pr}$ is a family of production morphisms (of type $p : (L \xrightarrow{r} R)$) indexed by production names, and and G_0 is the start graph of the grammar. Production names are used for identification purposes, so they should be unique. A direct derivation from a graph G_0 to a graph G_1 with $p_1 : L \rightarrow R$ at $m_1 : L \rightarrow G$ is denoted by $G_0 \overset{p_1, m_1}{\Longrightarrow} G_1$. A sequence of direct derivations of the form $G_0 \overset{p_1, m_1}{\Longrightarrow} \ldots \overset{p_k, m_k}{\Longrightarrow}{}^{*} G_k$ constitute a *derivation* from G_0 to G_k by $p_1, \ldots p_k$, briefly denoted by $G_0 \Rightarrow^* G_k$. The *graph language* generated by a graph grammar $\mathcal{G}$ is the set of all graphs G_k such that there is a derivation $G_0 \Rightarrow^* G_k$ using productions of $\mathcal{G}$ [Ehrig 1997].

Application and Consistency Conditions [Heckel 1995].
A *conditional constraint* $Cctr$ over a graph L is a pair consisting of a total morphism $x : L \to X$ and a set Θ of total morphisms $X \xrightarrow{y} Y$. A total morphism $L \xrightarrow{m} G$ satisfies a constraint $Cctr$ over L, denoted by $m \models_L Cctr$, if for all total injective morphisms $n : X \to G$ with $n \circ x = m$ there is a total injective morphism $o : Y \to G$ with $o \circ y = n$ for at least one $y \in \Theta$. A conditional *application condition* A_L over L is a set of conditional constraints. A total morphism $L \xrightarrow{m} G$ satisfies an A_L over L, denoted by $m \models_L A_L$ if $m \models_L Cctr$ for all $Cctr \in A_L$. Let $r : L \to R$ be a graph transformation rule. An application condition $A(r) = (A_L, A_R)$ for r is given by a *precondition* A_L and a postcondition A_R being conditional applications over L and R, respectively. Then $\hat{r} = (r, A(r))$ is called a *conditional rule*. A direct derivation $G \overset{r,m}{\Longrightarrow} H$ with co-match $R \xrightarrow{m^*} H$ is a direct conditional derivation $G \overset{\hat{r},m}{\Longrightarrow} H$ based on $\hat{r}$ if $m \models_L A_L$ and $m^* \models_R A_R$. Constraints in pre- and post-conditions of a rule are referred to as left- and right-sided constraints, respectively. A *consistency condition* $Ccnd$ is a set of total morphisms named *graphical consistency constraints*. A *consistency constraint* $c : P \to Q$ is satisfied by a graph G ($G \models c$) if for all total morphisms $p : P \to G$ there is a total morphism $q : Q \to G$ such that $q \circ c = p$. G satisfies $Ccnd$ ($G \models Ccnd$) if G satisfies all constraints $c \in Ccnd$.

Appendix B

Quality Sub-characteristics

Contents

B.1 Overview

Quality sub-characteristics follow the categorisation from the ISO/IEC 9126 standard and its corresponding update in the ISO/IEC 25000 standard series. Table B.1 identifies quality characteristics and sub-characteristics in the ISO/IEC 9126 quality model. A start symbol is added to new characteristics in the ISO/IEC 25010 standard (Software product Quality Requirements and Evaluation [SQuaRE] – Quality model). Emphasised in *italic* and **bold** are the targeted sub-characteristics in this investigation, they are derived from *functionality* and *maintainability* characteristics.

- *functionality* refers to the capability of a software system to provide functions which meet stated and implied needs when the system is used under specified conditions;
- *maintainability* refers to the capability of the software system to be modified – where modification may include corrections, improvements and functional specifications.

B.2 Suitability

Suitability refers to the adequacy of the software system in terms of its coverage of user needs and correctness of implementation. In the case of systems aiming at

Table B.1: Quality characteristics in ISO/IEC 9126 quality model

Quality Characteristic	Sub-characteristic
Functionality	***Suitability***
	Accuracy
	Interoperability
	Security
	Compliance
Reliability	Maturity
	Availability (∗)
	Fault tolerance
	Recoverability
	Compliance
Usability	Understandability
	Learneability
	Operability
	Attractiveness
	Compliance
	Technical accessibility (∗)
Efficiency	Time behaviour
	Resource utilisation
	Compliance
Maintainability	Modularity (∗)
	Reusability (∗)
	Analysability
	Changeability
	Stability
	Testeability
	Compliance
Portability	Adaptability
	Installability
	Co-existence
	Replaceability
	Compliance

integration of processes and applications, suitability focuses on coverage and correctness of integration needs. At the requirements and design stages, models are central. Integration needs are related to connected process steps that are supported by different systems, and hence requiring an integration effort. Process integration points are identified from data flows connecting process steps supported by different application architecture components. As several authors are pointed out as a factor of success [Puschmann 2004], [Janssen 2005], [Lam 2005], the assumption is that business integration precedes application integration. The suitability metric presented here follows a similar approach as previous metrics of functional adequacy and completeness regarding functional suitability [Brian 2006], [Bhatti 2005], [Gucegliogl u 2005], [Beaver 2005]. The principle behind is that lack of explicit knowledge of integration points can degrade suitability of an integration system design via functional incompleteness.

In this work, suitability is calculated as an estimated measure of functional completeness as is presented in ISO/IEC 9126.3. Equation B.1 refers to the suitability metric. An estimation of functional completeness-based suitability considers the number of documented process integration points (DIP) and the number of required process integration points (RIP). The different between DIPs and RIPs is that DIPs capture only process elements requiring integration that have an explicit relation to their supporting applications, instead RIPs only identify integration needs at process level without explicit connections to software levels.

The expression $\left(1 - \frac{RIP_i - DIP_i}{RIP_i}\right)$ in Equation B.1 indicates an estimated measure for the degree of functional completeness of the process-centric integration solution. A value closer to one would indicate an almost complete process-centric functional specification. This expression is adapted from the *functional implementation completeness* metric in the ISO/IEC TR 9126-3, which is calculated as $fic = 1 - (A/B)$, where A is the number of missing functions detected in an evaluation and B is the number of functions described in requirement specifications.

Suitability as in Equation B.1 also considers applications explicitly. It takes into account the ratio between the number of applications directly involved with process elements needing integration ($\sum_j A_j(RIP_i)$) and the total number of applications involved in the whole scenario ($\{A\}$). In general, an integration scenario can span several processes in different organisations, but it can also be framed within a single organisation. In that case, $\{A\}$ would account for all applications spanning the enterprise-application landscape. The ratio $\frac{\sum_j A_j(RIP_i)}{\{A\}}$ scales the estimated measure of functional completeness described above according to the importance of each integration point. The larger the number of involved applications at an integration point, the greater the influence over the suitability measure. If an integration point involved all applications in the scenario, the ratio would equal to one.

$$suitability = \sum_i \left[\left(1 - \frac{RIP_i - DIP_i}{RIP_i}\right) \cdot \frac{\sum_j A_j(RIP_i)}{\{A\}} \right] \tag{B.1}$$

where i indicates a control flow connector from a process requiring integration, and j an specific application connected to process elements from an integration point.

B.3 Functional Compliance

Considering functional characteristics of a software, compliance refers to the degree to which the software product adheres to standards, conventions or regulations in laws and similar prescriptions relating to functional suitability.

For process-centric integration systems, an important aspect of functional com-

pliance is compliance to process regulations [Daniel 2009], [Kharbili 2008]. The capabilities to identify non-compliance at process levels can indicate failures from the business point of view [Lu 2008] and therefore benefit analysability characteristics of a process-centric system.

A number of approaches have indicated the benefits of checking process-level compliance through the identification of regulatory process structures (or process compliance patterns [Ghose 2008]). Similar to how design and architectural patterns constrain software architectures, these process patterns would act as regulatory constraints on concrete process models.

B.4 Maintainability: Changeability and Analysability

Organisations are in disadvantage if their capacity to change quickly and reliably is low. This include changes on their software systems. Software systems no longer evolve as separate entities but evolve integrated with each other in a complex interrelated system [Land 2003]. Maintainability characteristics of integration solutions are central to assess an organisation's capacity to change.

Generally speaking, maintainability is associated to the effort required to make specified modifications to a software system. Changeability and analysability are two sub-characteristics of maintainability.

- *Changeability* refers to the ease of changing the system and
- *Analysability* refers to the ease of understanding the system design.

Classic metrics for software maintainability have focused on code levels [Coleman 1994], however maintenance activities for process-centric integration systems concentrate a significant initial effort on analysing the involved architecture and process models. This is especially true with the increasing adoption of Model-driven Development (MDD) [Rech 2009]. Even though the abstraction level has increased with the advent of MDD, models continue to change, and hence, they would need to be constantly analysed. In that context, *analysability* becomes relevant and observations regarding *changeability* should shift the focus towards model levels.

B.4.1 Changeability

Changeability refers to the degree to which the product enables a specified modification to be implemented, more precisely, it refers to the ease with which a software product can be modified to meet specific requirements. Implementation can include coding, designing and documenting changes. Changes can also include installation of updates and upgrades. Analysability can influence changeability.

During requirements and design stages of an integration project, *changeability* can be related to changes on models containing the processes and applications to be integrated. To consider changeability at these stages, the quantification approach proposed in [Ross 2008b] is utilised. This approach is generic to large scale systems and considers quantitative metrics of derived characteristics related to changeability. In [Ross 2008b], the authors quantify *changeability* in relation to three aspects: change agents, change effects and change mechanisms. According to change effects, derived quality characteristics include *modifiability*, *scalability* and *robustness*. Because the main interest here is on change effects (due to their implications on maintenance costs) a revision of those concepts is provided below.

B.4.1.1 Modifiability

Modifiability of a system design i in one of its attributes $m \in \{X\}_i$ is calculated as the number of possible paths allowing the transit from the design i to possible designs that have the attribute m added to or subtracted from its attribute set $\{X\}_i$, but whose transition mechanism cost less than an acceptability threshold $\hat{C}$. Equation B.2 indicates modifiability of design i to adding or substracting attribute X^m.

$$modifiability_i^m : \sum_j [F_{X^M}(\{DV\}_i) \cap F_{X^M}(\{DV\}_j) = X^m, T_{ijk} < \hat{C}, \forall j \in S, k \in R] \quad \text{(B.2)}$$

with $F_{X^M} : \{DV^N\} \rightarrow \{X^M\}$ and $f_C : \{DV^N\} \rightarrow C$; $f_U : \{X^M\} \rightarrow U$, and

- f_C : design variable to cost mapping function.
- DV^N : set of N design variables.
- C : cost (it can be instantiated through cost models, e.g. as suggested in [Themistocleous 2004]).
- f_U : attribute to utility mapping function (for instance, considering [Wierzbicki 1982]).
- X^M : set of M attributes (it corresponds to the parametrisation of perceived value).
- T_{ijk} : accessibility tensor of size $R \times S \times S$. The ijk entry identifies the transition cost from design y ($\{DV\}_i$) to design j ($\{DV\}_j$) by means of transition rule k.
- S : set of designs.
- R : set of transition rules.

The transition from a design i to a design j is through the mechanism k and it is captured by the accessibility tensor T_{ijk}. Designs and design transitions belong to

a tradespace network capturing the representations of designer-controlled technical parameters ($\{DV\}$, having a measurable *cost*) and stakeholder-perceived value parameters ($\{X\}$, having a measurable *utility*). For instance, a stakeholder-perceived value parameter for an integration system could be its capability for reviewing security credentials of users sending messages in particular processes. A designer-controlled technical parameter associated to that stakeholder-perceived value parameter can be associated to the verification of digital signatures of SOAP messages signed using X.509 certificates. This stakeholder-perceived value parameter (associated to a *utility* value) versus designer-controlled technical parameter (associated to a *cost*) defines a point in a tradespace network that relates *cost* to *utility* points.

In an scenario of service-based integration systems, services can be used as intermediary architecture elements to abstract process and application integration levels. An additional level of indirection can generate additional costs of implementation and maintenance. However, given the potential of services to be reused at different integration points, service-based integration systems can potentially reduce costs of operation and maintenance (including modifications) in the medium and long term, especially for large systems [Erl 2004]. However, if the design of the service-based system involves services with functional redundancy, benefits can be reduced by increased cost of development and maintenance [Papazoglou 2007]. In this work, services with functional redundancy are identified through the identification of similar process structures being supported by different services. This is also an indication of possible redundant application support and it represents an opportunity to software level component rationalisation.

B.4.1.2 Scalability

Scalability refers to the capability of a system to increase the value of its attributes. Scalability of a design i to raising or lowering the value of attribute X^m can be calculated using Equation B.3. Scalability is only discussed in the case study of Section 7.2. Further evaluation is not within the scope of this work.

$$scalability_i^m : \sum_j [[F_{X^M}(\{DV\}_i)]^m - [F_{X^M}(\{DV\}_j)]^m \neq 0, T_{ijk} < \hat{C}, \forall j \in S, k \in R] \quad \text{(B.3)}$$

B.4.1.3 Robustness

Robustness refers to the capability of a system to be insensitive under changing conditions regarding its perceived value. A system whose attributes can satisfy new requirements created over time is considered value robust [Ross 2008a]. This characteristic can be achieved through passive or active means. Active value robustness can

be achieved by designs with increased changeability and accessibility. Passive value robustness can be achieved by systems that may have excess capability or a large set of latent value, thus increasing the likelihood of satisfying new requirements without changing the system [Ross 2009]. Passive value robustness can be assessed by evaluating the system's capabilities to satisfy desired attributes in different scenarios. To instantiate this type of evaluation, the authors in [Ross 2009] suggest the use of epoch-era analysis. In this type of analysis the system's lifetime is divided into epochs, which define scenarios that span a determined time when significant design characteristics, requirements and context variables are fixed. Multiple sequentially related epochs can identify an era, which can capture a longer period on the system's lifetime.

B.4.2 Analysability

Analysability refers to the effort or resources spent when a system requires diagnosis or analysis during maintenance activities. Analysability metrics should provide a medium to predict and measure the spent effort or spent resources in trying to diagnose for deficiencies or causes of failure, or for identification of parts to be modified in a software. Enterprise (processes and applications) integration solutions cover different abstraction layers, including processes, services and applications. Metrics to measure analysability of an integration system should cover all of these layers and their interconnections.

Additional to metrics directly associated to characteristics and subcharacteristics, there are pure internal metrics[1] that can be used to measure certain attributes of the software design (and code) that can influence all of the overall software characteristics. Pure internal metrics that can be used to indirectly measure analysability, and also maintainability, are the metrics of *traceability* and *complexity*.

B.4.2.1 Traceability

Traceability, as defined in the ISO/IEC 9126-3, aims to provide a medium to measure the effectiveness of documentation and design structure and code of software product in mapping functions from requirements to implementation. However, traceability in an integration solution, which involves different layers, requires support to trace dependencies between elements in different layers, specially when changes need to take place. For instance, if an application is being updated, it is necessary to know what other applications and what processes are being affected. A similar situation occurs when changes at process level can influence the modification

[1] Review annex in ISO/IEC TR 9126-3 regarding pure internal metrics.

of the underlying software architecture (including software services and application components).

Equation B.4 is a simple measure of traceability based on the pure internal metric of traceability from ISO/IEC 9126-3. It indicates the ratio between effectively traced elements in a model ($\sum_i ETE_i$) and the total number of traceable elements for such a model ($\sum_i TE_i$). The closer the metric is to one, the better. Note that even though more information regarding dependencies between different elements of the integration problem and its changes over time can be beneficial, this would normally imply an increased cost of maintaining traces (or trace links). But, as almost the norm in any system design, tradeoffs between different characteristics should be balanced according to the needs.

$$traceability = \frac{\sum_i ETE_i}{\sum_i TE_i} \tag{B.4}$$

Traceability support can greatly benefit analysability but to the cost of maintaining trace links. The argument to invest in trace links maintenance is that they could save greater costs generated during modifications to large enterprise processes and applications integration systems. The risk of poor requirements satisfaction due to insufficient or incorrect analysis is likely when there is no awareness of the impact of changes in related elements from other layers [Lankhorst 2005], [Land 2007].

B.4.2.2 Complexity

Beside computational complexity that allows to classify problems addressed by software according to its difficulty to be solved and resources utilised, software complexity refers to the characteristics of a software that make it easier or more difficult to understand and maintain.

As mentioned previously, the design of process-centric integration systems based on services covers different abstraction layers. For software architecture layers, it is proposed that large systems with a regular substructure are simple to create and maintain, whereas systems created in an ad hoc fashion quickly become unmaintainable [Kazman 1998]. Since measures of coupling and cohesion, or fan-in and fan-out have been considered less reliable to be correlated with architectural complexity, measures of *architectural regularity* have been used to assess architecture complexity. In particular, metrics for architecture's pattern coverage have been utilised to investigate critical design aspects of architectures (for instance, occurrence of cyclic patterns such as *mutual recursion*). For process and process-centric service layers, several measures of complexity have been proposed in analogy to metrics of software complexity such as LOC (Lines Of Code), cyclomatic complexity, control-flow complexity, and Halstead-based metrics, among others [Gucegliogl 2005], [Cardoso 2006],

[Vanderfeesten 2007]. However, these metrics in the software context have been criticised, for example, for being not sensitive enough to rank different software or because the amount of time required to create software code is difficult to predict [Weyuker 1988]. Following the proposal in [Kazman 1998] and suggestions/evidences in [Gruhn 2006], [Mendling 2007], [Hirzalla 2009], here complexity of process-centric service architectures and process models is assessed through measuring coverage of structural regularities in model and architecture descriptions, in particular, pattern coverage.

During initial stages of process and application integration system development, analysability characteristics regarding architecture and process models influence the capacity to assess the complexity of systems. A lower capacity to assess complexity could increase the effort of analysing the system's design and future modifications.

B.4.2.3 On the relation between modifiability and analysability

Consider a modification to a enterprise process and application integration system in an attribute m is required. Process integration points not linked (*traced*) to application components would require – in the worst case – a revision of the entire enterprise-wide application architecture model in order to find possible ways to transit from an original design i to a new design j. The cost of analysing a change at model level would be increased and; therefore, *analysability* would be negatively affected. The cost would vary from the cost of reviewing only application components (Ac) related to attribute m in design i, to the cost of reviewing all architecture components in the enterprise-wide (or process-wide) application architecture.

B.5 Reusability

According to the standard ISO/IEC 25010, reusability refers to the degree to which an asset can be used in more than one software system, or in building other assets.

Reusability has been considered one of the basic principles in service-oriented architectures. When considering process and application integration systems using services as intermediary elements to mediate between process and application integration layers, reuse of services can have at least two connotations. These two refer to reuse at business and software levels. As pointed out in several integration case studies such as [Puschmann 2004], [Janssen 2005] and [Lam 2005], business process integration should precede technology integration. Considering this evidence, even though the application architecture level is a relevant part of the approach proposed in this work, the developed techniques concentrate efforts on business process and process-centric service levels.

On the other hand, for large and distributed organisations, a common situation in practice is that software systems created to solve a similar problem, but in different times and locations, often do not share their design solutions. Design knowledge reuse and utilisation of architectural abstractions like patterns have been considered central to reduce the complexity of architecture designs and to facilitate their analysis [Buschmann 2007] [Daniel 2009], thus improving maintainability.

Focusing on process patterns, reusability would be assessed through the capacity to identify sections in concrete process models where instances of patterns can be identified and defined as potential new reusable services (see Section 3.3.2 for details). A metric for reusability may need more than a simple counting of how many times a service was reused. For example, while a payment service is likely to be reused within particular business domains (e.g., e-commerce or financial domains), an authentication service could be used (broadly) in several business domains, but in a specific technical domain (such as security). Thus, a metric to assess reusability of services would need the definition of a certain context where the services are being reused. Considering a service defined on the basis of a process pattern, Equation B.5 describes a metric to estimate the potential reuse of a service s_i within the context of a particular process P_j, where s_k indicates an instance of service utilisation. The greater is the reusability measure, the better. In an extreme, if all utilised services in P_j are s_i, the metric is equal to one.

$$reusability_i^j = \frac{\sum_k [s_k | s_k = s_i, s_k \subset P_j]}{\sum_k [s_k | s_k \subset P_j]} \tag{B.5}$$

Appendix C

Complementary Information for Case Studies

Contents

C.1 EBPP Case Study's Complementary Information

C.1.1 Business and Application Level Models

This section illustrates models used in the EBPP case study in Section 7.2.3. Models refer to the main utility company and they follow the guidelines suggested for the LABAS's BAIL layer. Process level elements are connected through trace links with business model elements and application architecture elements. Subsequent steps involved in the framework generate new models that add information regarding new services. Services are the building blocks of a service-based architectural solution to integrate the EBPP's processes and applications. Figure C.1 shows the high level EBPP process with associated domain model and application architecture elements. The EBPP process is disaggregated and presented in a number of figures as follows:

- *banks network*: Figure C.2 shows the transfer money activity performed by the *banks network* role and associated domain model and application architecture elements. The role is further divided into the utility company's bank, the client's bank and an intermediary role corresponding to a clearing house.
- *utility company*: Figures C.3 to C.8 show activities, tasks, domain model and application architecture elements supporting activities performed by the *utility company* role (i.e., the *generate bill*, *send bill*, *liquidate debt* and *accumulate debt* activities).
  - Two activities from *generate bill* activity, which are *get customer number* and *get current debt*, are further decomposed in Figures C.5 and C.4.
  - In turn, the activity *get customer consumption* from *get current debt* is shown in Figure C.6.

- The *send bill* activity is performed by a *customer service provider* role on behalf of the *utility company* role.

- *customer*: the activities receive bill and pay bill performed by the customer role are not detailed since their relation to domain model and application architecture elements is illustrated in Figure C.1.

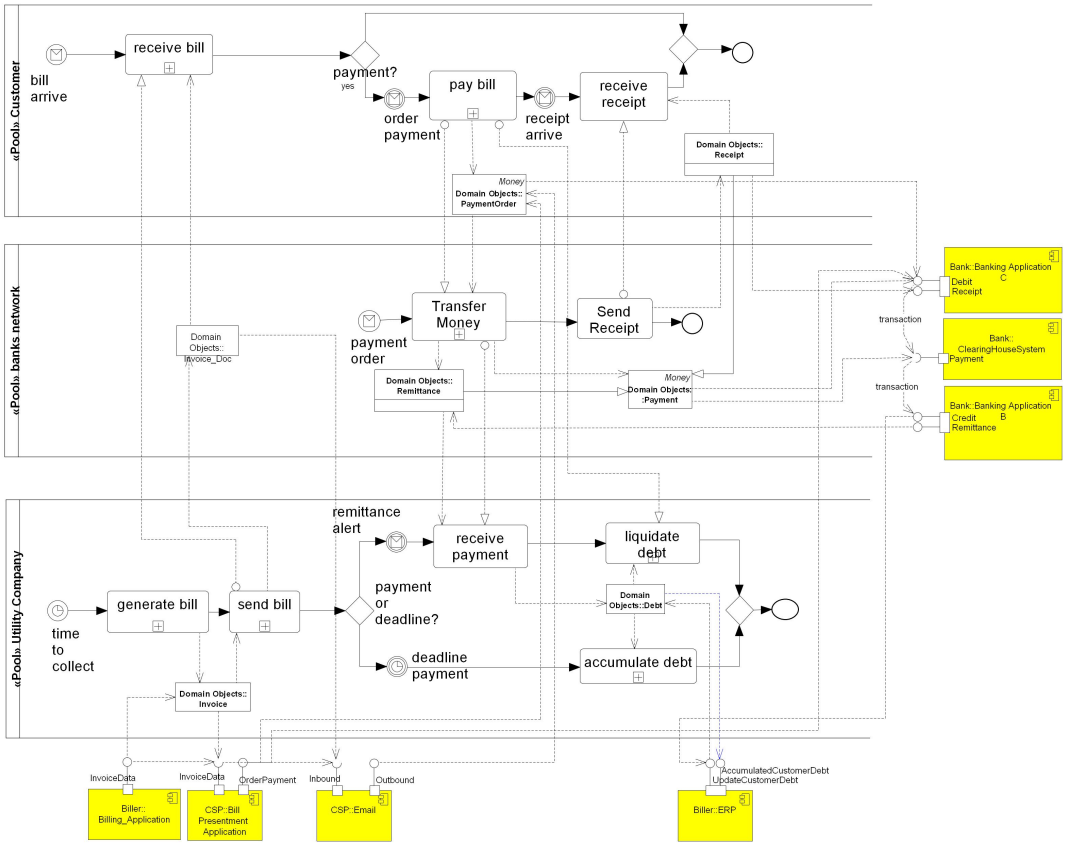


Figure C.1: EBPP process from Figure 7.8 in BAIL layer.

Two additional figures, Figure C.9 and C.10, illustrate the *generate bill* activity in the two new subsidiary companies of the main utility company.

C.1.2 Intermediary Services between Business and Application Level Models

This section adds information from the service architecture to BAIL layer models. These models are derived from architecture decisions in the second scenario of the EBPP case study.

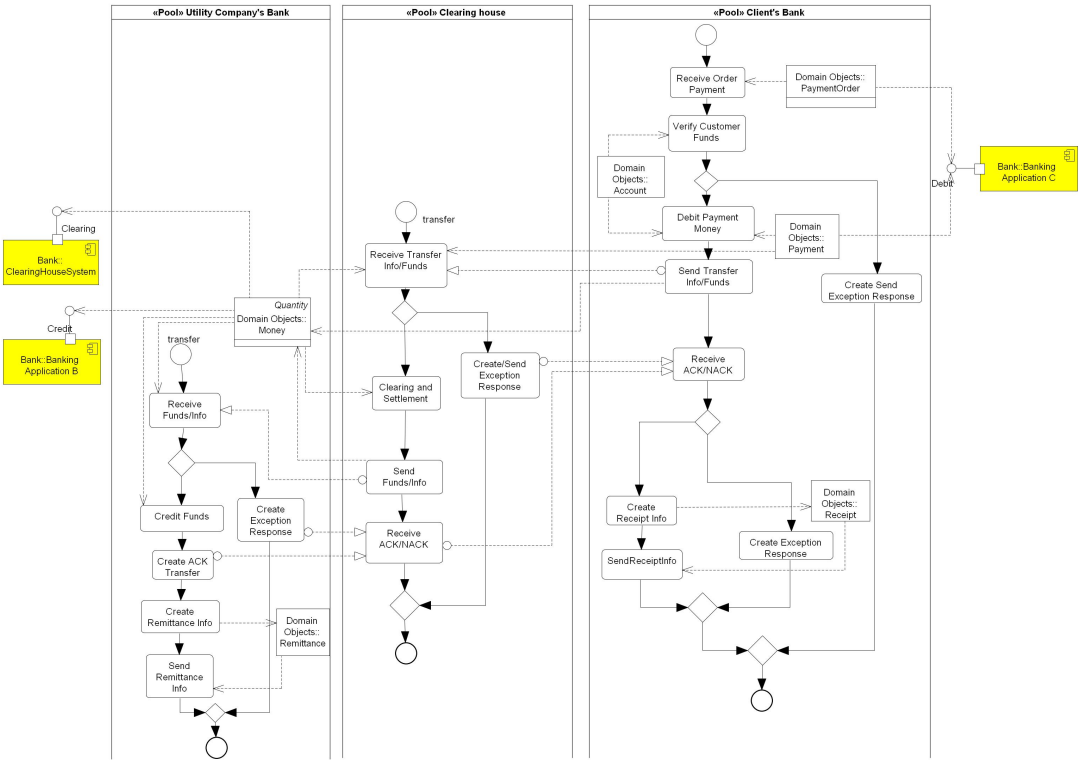


Figure C.2: *transfer money* activity in BAIL layer.

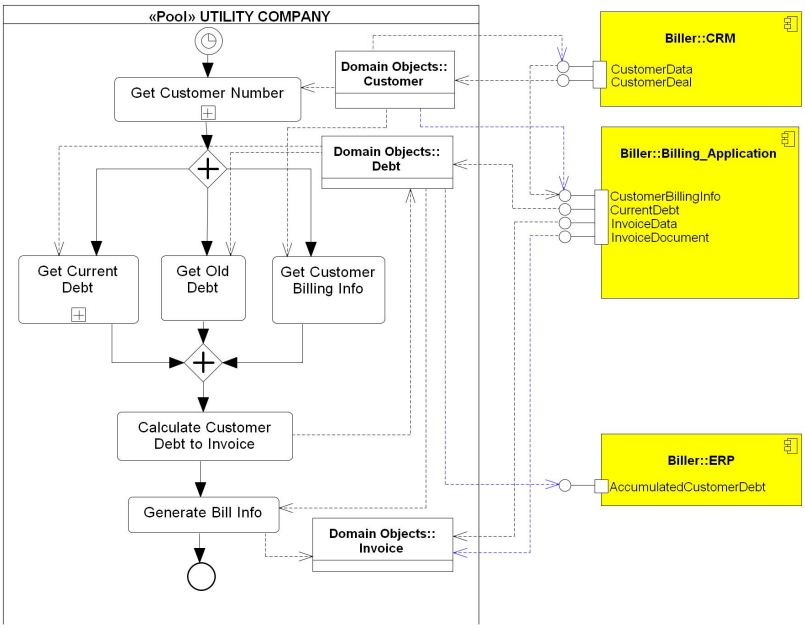


Figure C.3: *generate bill* activity in BAIL layer.

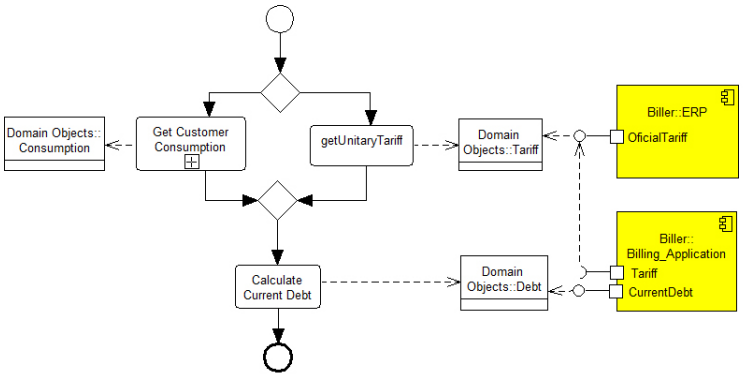


Figure C.4: *get current debt* activity in BAIL layer.

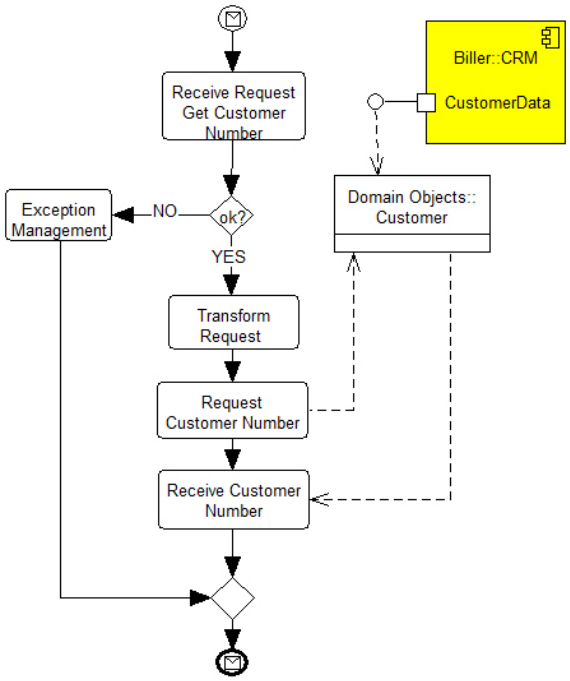


Figure C.5: *get customer number* activity in BAIL layer.

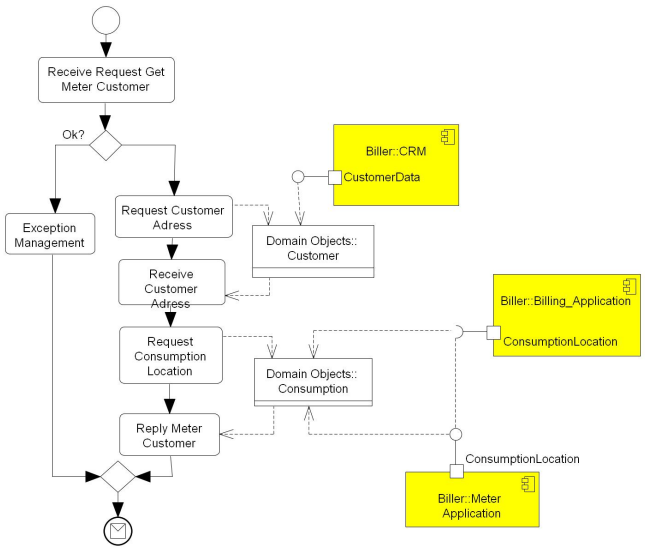


Figure C.6: *get customer consumption* activity in BAIL layer.

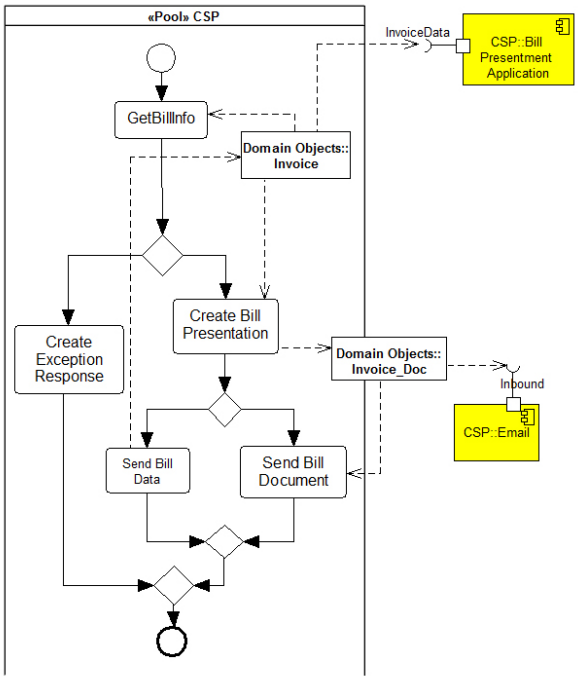


Figure C.7: *send bill* activity (*customer service provide* role on behalf of main *utility company*) in BAIL layer.

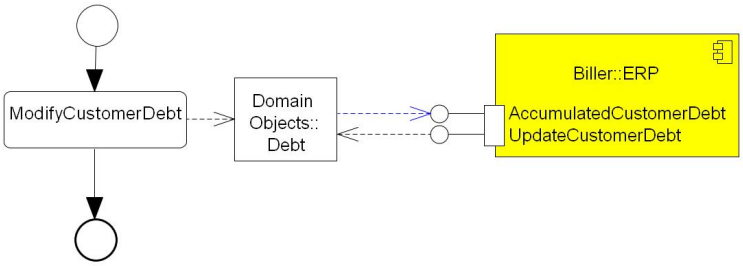


Figure C.8: *accumulate and liquidate debt* activities in BAIL layer.

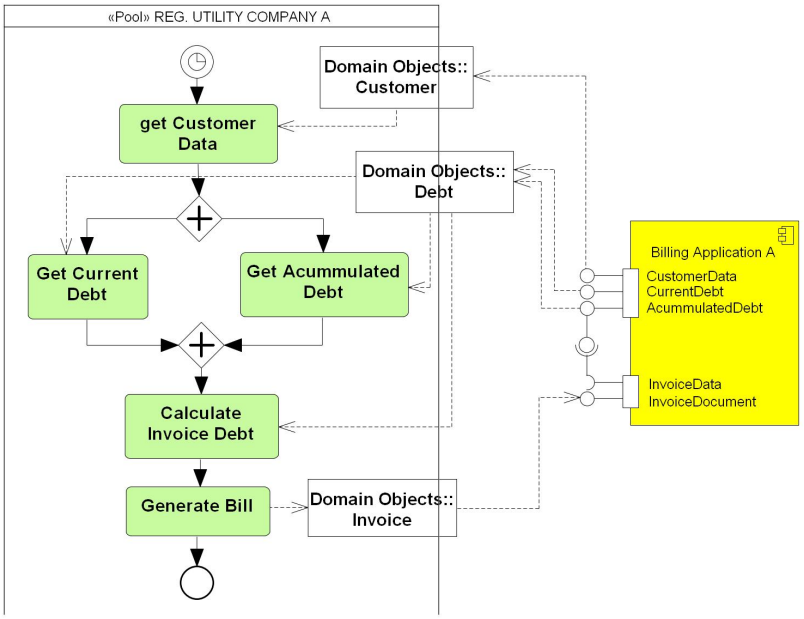


Figure C.9: *generate bill* activity in BAIL layer for *Utility Company A*.

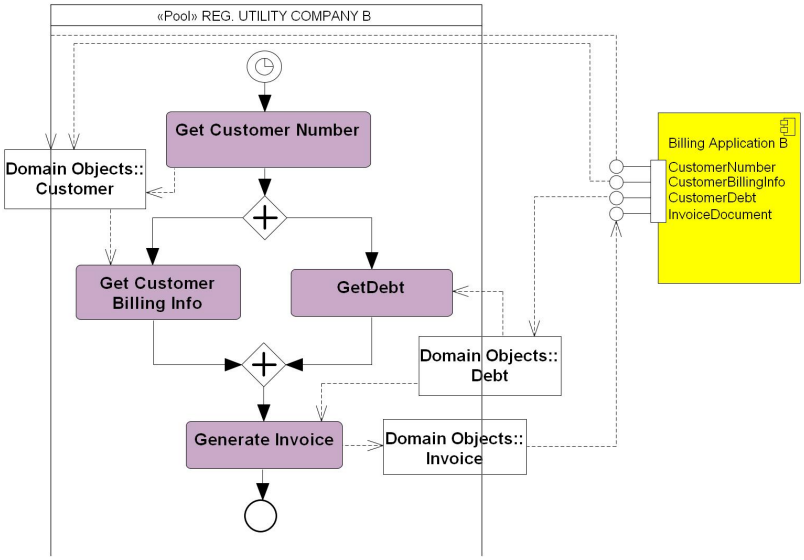


Figure C.10: *generate bill* activity in BAIL layer for *Utility Company B*.

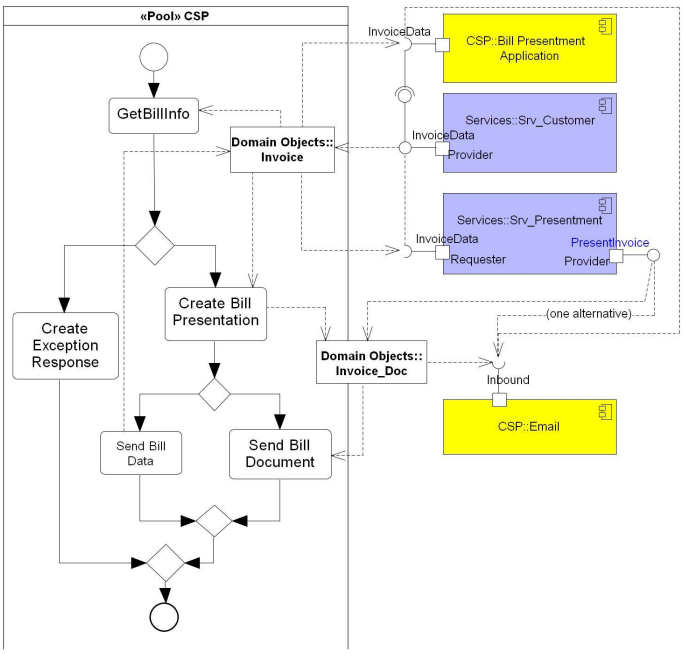


Figure C.11: *send bill* activity (*customer service provide* role on behalf of main *utility company*) in BAIL layer and associated services.

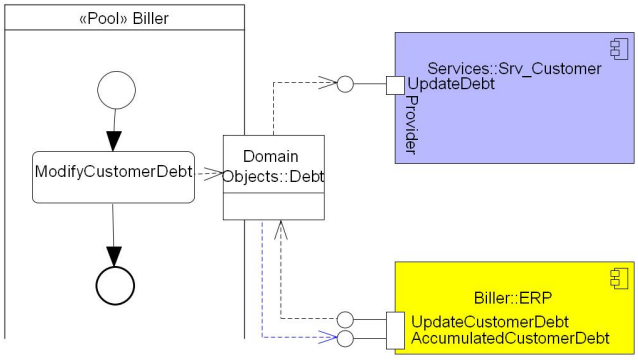


Figure C.12: *accumulate and liquidate debt* activities in BAIL layer and associated services.

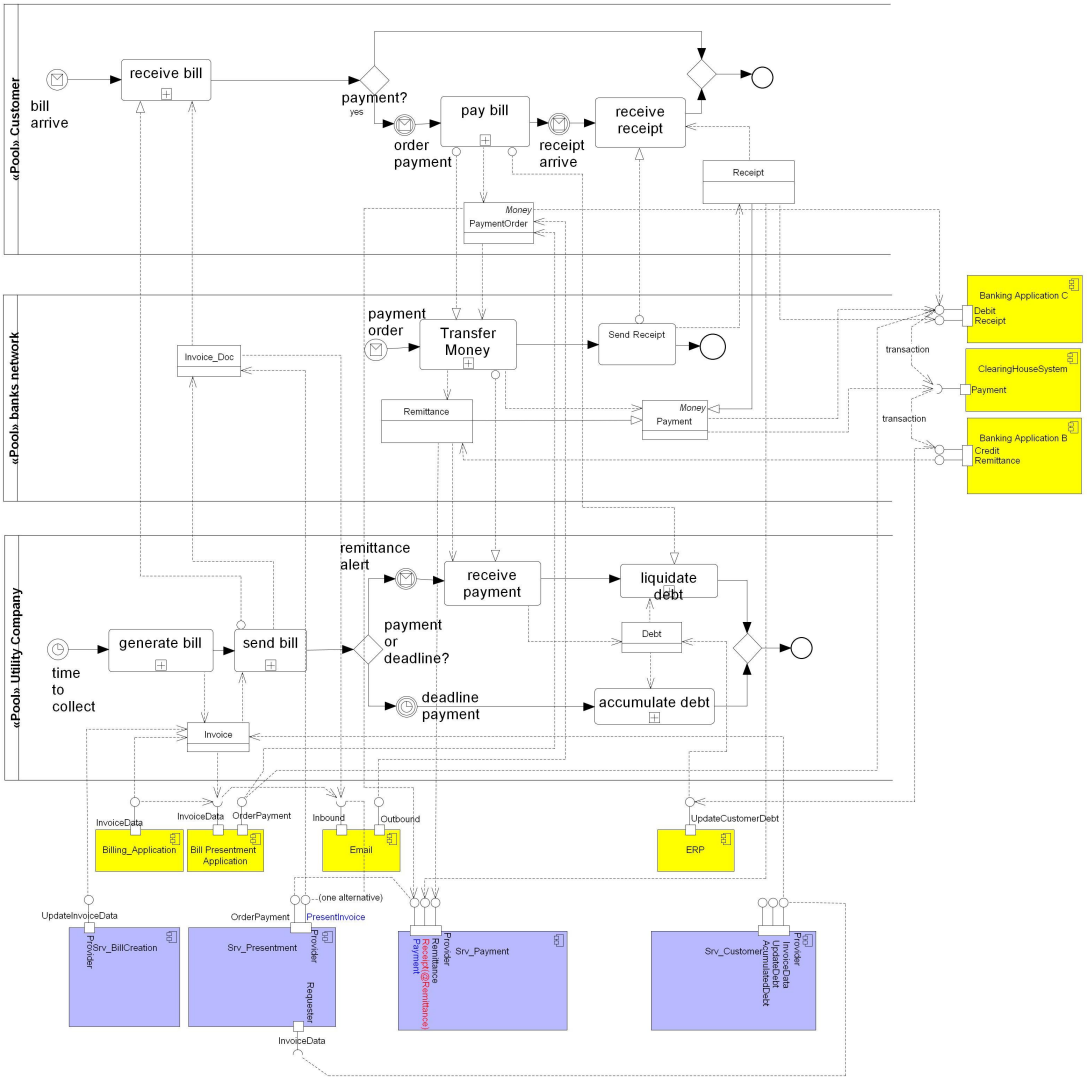


Figure C.13: EBPP process from Figure 7.8 in BAIL layer and associated services.

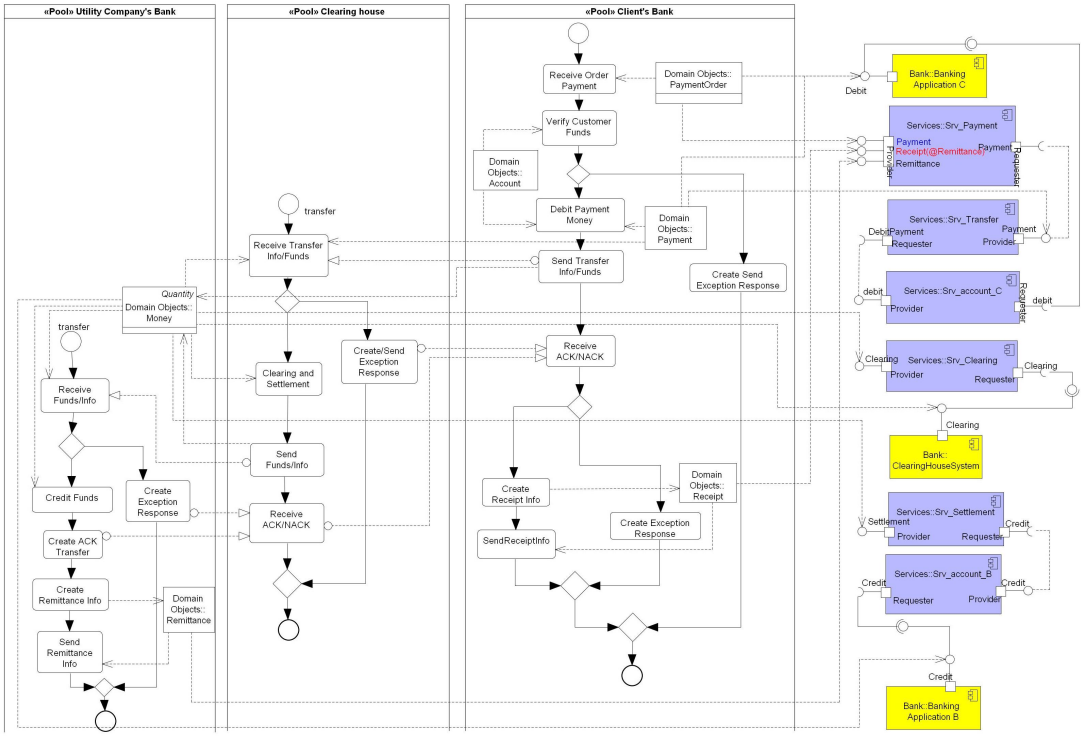


Figure C.14: *transfer money* activity in BAIL layer and associated services.

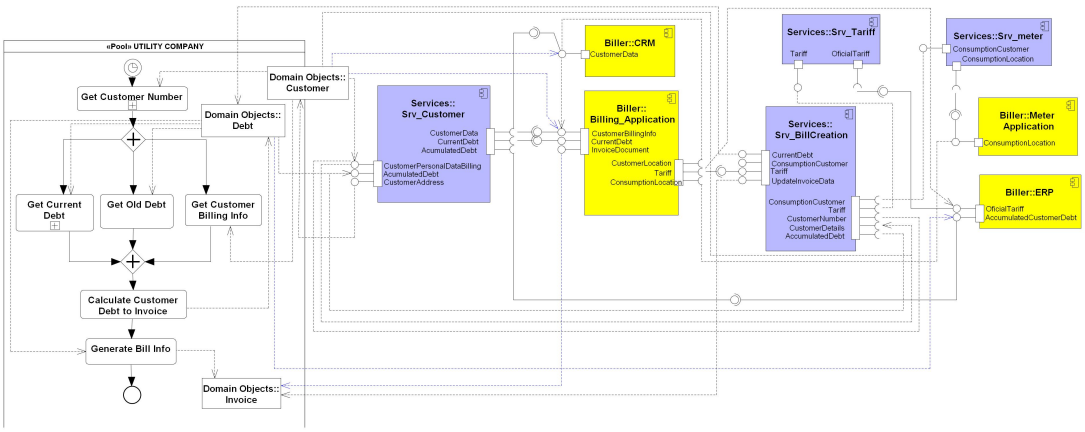


Figure C.15: *generate bill* activity in BAIL layer and associated services.

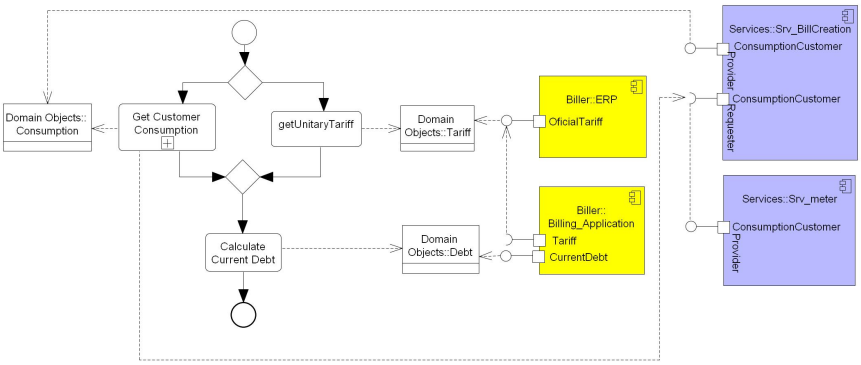


Figure C.16: *get current debt* activity in BAIL layer and associated services.

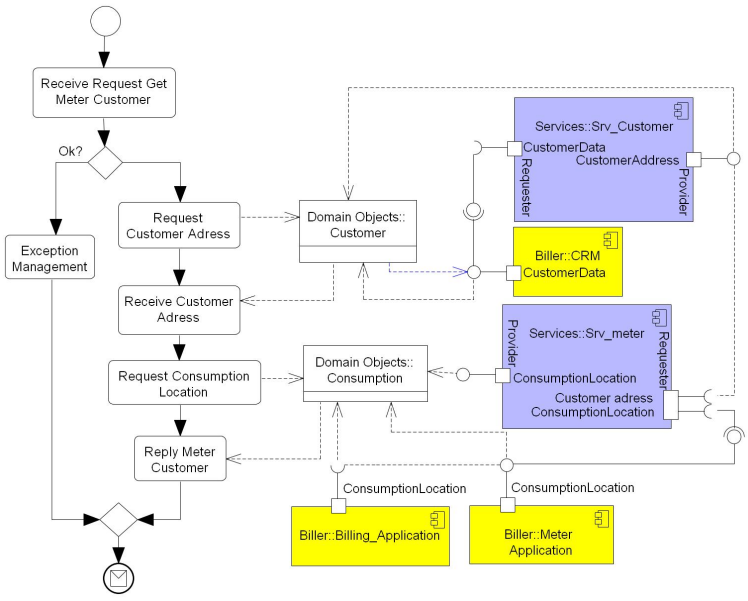


Figure C.17: *get customer consumption* activity in BAIL layer and associated services.

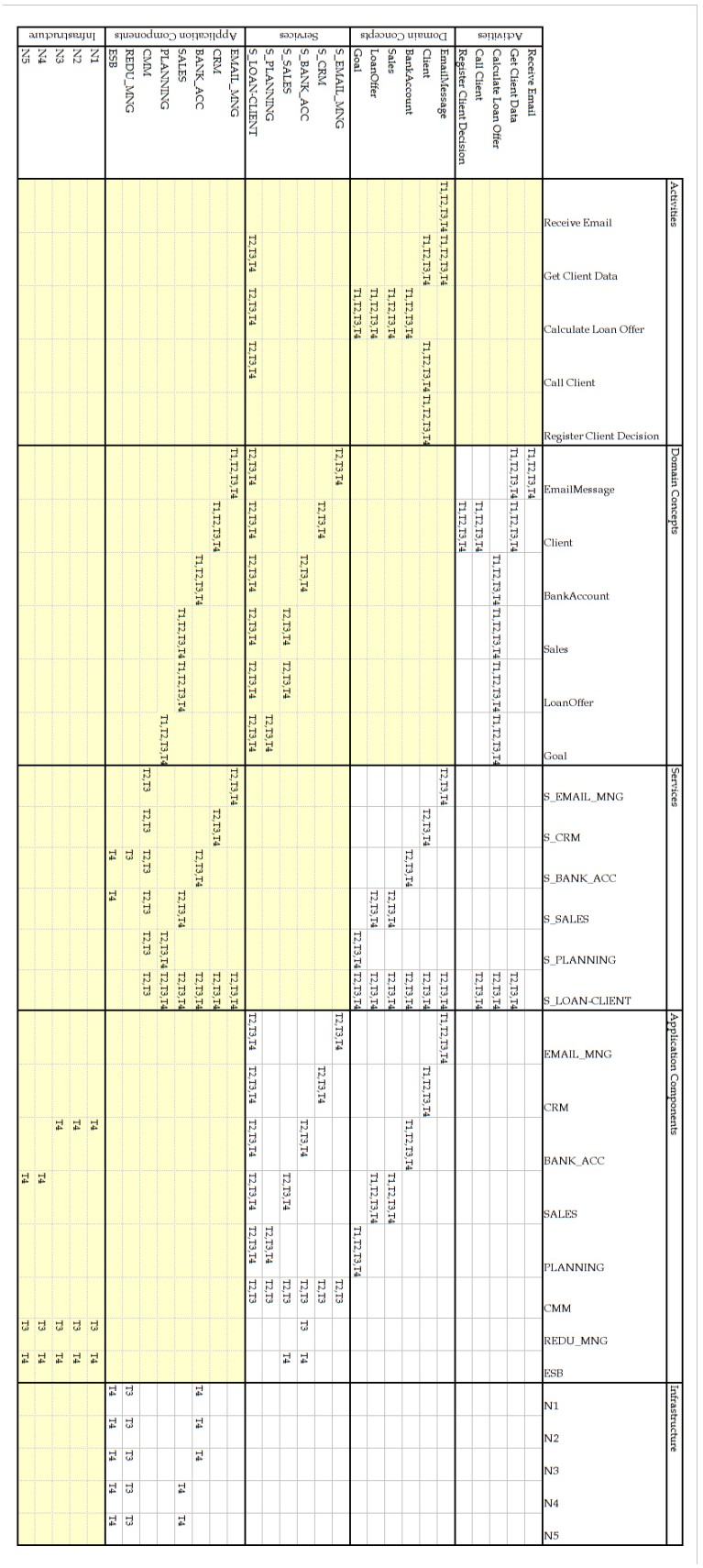
C.2 LM Case Study's Complementary Information

Trace Links. The main model-to-model trace links for elements in different layers involving the LM process (from the point of view of the *phone sale agent* role from Figure 7.1) and associated domain model elements and architecture components are indicated in Table C.1. Note that if there is a trace link connecting two elements in one of the technical scenarios described in Section 7.2.2.3, for example the first scenario, then a T1 is annotated in the cell relating those two elements in the table. Similarly, T2, T3 and T4 are used to denote the existence of a trace link in subsequent scenarios: two, three and four.

Table C.2 provides a reference of the existing trace links between pattern roles and model elements. The columns in the table show the pattern names (first row), pattern roles (second row) and the scenarios described in the previous section (T1 to T4 in the third row). Different pattern instances are identified as P1 to P16 in the table. Note that a pattern role might be fulfilled by different model elements in different scenarios. For instance, the message coordinator role in the reliable messaging pattern [Erl 2008] is fulfilled in the second and third scenarios (T2, T3) by the CMM component, and by the ESB component in the fourth scenario (T4) in the same pattern instance – P14 in Table C.2.

Figure C.18 illustrates one implementation of the *Loan to Client* pattern as a service based on functionality available from existing applications.

Table C.1: Model-to-model Trace Links Reference.



Activities	Domain Concepts	Services	Application Components	Infrastructure
Receive Email	EmailMessage	S_EMAIL_MNG	EMAIL_MNG	N1
Get Client Data	Client	S_CRM	CRM	N2
Calculate Loan Offer	BankAccount	S_BANK_ACC	BANK_ACC	N3
Call Client	Sales	S_SALES	SALES	N4
Register Client Decision	LoanOffer	S_PLANNING	PLANNING	N5
	Goal	S_LOAN-CLIENT	CMM	
			REDU_MNG	
			ESB	

Activities		
	Receive Email	T1,T2,T3,T4
	Get Client Data	T1,T2,T3,T4
	Calculate Loan Offer	T1,T2,T3,T4
	Call Client	T1,T2,T3,T4
	Register Client Decision	T1,T2,T3,T4
Domain Concepts		
	EmailMessage	T1,T2,T3,T4
	Client	T1,T2,T3,T4
	BankAccount	T1,T2,T3,T4
	Sales	T1,T2,T3,T4
	LoanOffer	T1,T2,T3,T4
	Goal	T1,T2,T3,T4
Services		
	S_EMAIL_MNG	T2,T3,T4
	S_CRM	T2,T3,T4
	S_BANK_ACC	T2,T3,T4
	S_SALES	T2,T3,T4
	S_PLANNING	T2,T3,T4
	S_LOAN-CLIENT	T2,T3,T4
Application Components		
	EMAIL_MNG	T1,T2,T3,T4
	CRM	T1,T2,T3,T4
	BANK_ACC	T1,T2,T3,T4
	SALES	T1,T2,T3,T4
	PLANNING	T1,T2,T3,T4
	CMM	T2,T3
	REDU_MNG	T3
	ESB	T4
Infrastructure		
	N1	T4
	N2	T4
	N3	T4
	N4	T4
	N5	T4

Table C.2: Model-to-pattern Trace Links Reference.

		RPI-PAT											
		ApplicationA				Invoked Function				ApplicationB			
	Scenarios	T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4
Pool	Phone sales agent												
	* GetKeyManager(CRMReq) : key_mng												
	* CallClient(name_client, phone)												
	Client												
	Direct sales agent												
	* OfferLoanContract												
	* ReportSupervisor												
	* SignVisitClientDoc												
Services	S_EMAIL_MNG												
	* GetMsgDetails(IdQ) : date_msg, email_add												
	S_CRM												
	* CustName(email_address) : name_cust												
	* CustDetails(key_mng, name_cust) : address_cust, phone, email_cust												
	S_BANK_ACC												
	* ClientName(email_address) : name_client												
	* FundsAcc(ID_client, account_num) : funds_account												
	* IDClient(name_client) : ID_client												
	* AccNumber(ID_client) : account_num												
	S_SALES												
	* CalculateOffer(delta_goal, funds_account) : offer_client												
	* SalesQuarter(Quarter) : salesQuarter												
	S_PLANNING												
	* CalculateDeltaGoal(goalQuarter, salesQuarter) : delta_Goal												
	S_LOAN-CLIENT												
	* OfferLoanClient (email_adress) : offer_client												
Application Components	EMAIL_MNG												
	* MsgDetails(IdQ) : date_msg, email_add												
	CRM												
	* Get_CustName(email_address) : name_cust												
	* Get_CustDetails(key_mng, name_cust) : address_cust, phone, email_cust												
	BANK_ACC												
	* Get_ClientName(email_address) : name_client												
	* Get_FundsAcc(ID_client, account_num) : funds_account												
	* Get_IDClient(name_client) : ID_client												
	* Get_AccNumber(ID_client) : account_num												
	SALES	P1											
	* CalculateOffer(delta_goal, funds_account) : offer_client												
	* Get_SalesQuarter(Quarter) : salesQuarter					P1							
	PLANNING									P1			
	* CalculateDeltaGoal(goalQuarter, salesQuarter) : delta_Goal					P1							
	REDU_MNG												
	CMM												
	ESB												
Infrastructure	N1												
	N2												
	N3												
	N4												
	N5												

		LW-PAT																			
		Wrapped Applic.				Applic. Operation				Service				Offered Functionality				Required Service			
	Scenarios	T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4
Pool	Phone sales agent																				
	* GetKeyManager(CRMReq) : key_mng																		P3	P3	P3
	* CallClient(name_client, phone)																				
	Client																				
	Direct sales agent																				
	* OfferLoanContract																				
	* ReportSupervisor																				
	* SignVisitClientDoc																				
Services	S_EMAIL_MNG										P2	P2	P2						P3 to P5	P3 to P5	P3 to P5
	* GetMsgDetails(IdQ) : date_msg, email_add														P2	P2	P2				
	S_CRM										P3	P3	P3								
	* CustName(email_address) : name_cust														P3	P3	P3				
	* CustDetails(key_mng, name_cust) : address_cust, phone, email_cust														P3	P3	P3				
	S_BANK_ACC										P4	P4	P4						P5	P5	P5
	* ClientName(email_address) : name_client														P4	P4	P4				
	* FundsAcc(ID_client, account_num) : funds_account														P4	P4	P4				
	* IDClient(name_client) : ID_client														P4	P4	P4				
	* AccNumber(ID_client) : account_num																				
	S_SALES										P5	P5	P5						P6	P6	P6
	* CalculateOffer(delta_goal, funds_account) : offer_client														P5	P5	P5				
	* SalesQuarter(Quarter) : salesQuarter														P5	P5	P5				
	S_PLANNING										P6	P6	P6						P5	P5	P5
	* CalculateDeltaGoal(goalQuarter, salesQuarter) : delta_Goal														P6	P6	P6				
	S_LOAN-CLIENT										P7	P7	P7						P2 to P6	P2 to P6	P2 to P6
	* OfferLoanClient (email_adress) : offer_client														P7	P7	P7				
Application Components	EMAIL_MNG		P2	P2	P2																
	* MsgDetails(IdQ) : date_msg, email_add						P2	P2	P2												
	CRM		P3	P3	P3																
	* Get_CustName(email_address) : name_cust						P3	P3	P3												
	* Get_CustDetails(key_mng, name_cust) : address_cust, phone, email_cust						P3	P3	P3												
	BANK_ACC		P4	P4	P4																
	* Get_ClientName(email_address) : name_client						P4	P4	P4												
	* Get_FundsAcc(ID_client, account_num) : funds_account						P4	P4	P4												
	* Get_IDClient(name_client) : ID_client						P4	P4	P4												
	* Get_AccNumber(ID_client) : account_num																				
	SALES		P5	P5	P5																
	* CalculateOffer(delta_goal, funds_account) : offer_client						P5	P5	P5												
	* Get_SalesQuarter(Quarter) : salesQuarter						P5	P5	P5												
	PLANNING		P6	P6	P6																
	* CalculateDeltaGoal(goalQuarter, salesQuarter) : delta_Goal						P6	P6	P6												
	REDU_MNG																				
	CMM																				
	ESB																				
Infrastructure	N1																				
	N2																				
	N3																				
	N4																				
	N5																				

		SM-PAT								RM-PAT								RI-PAT											
		Service				Supported Service operation				Message Coordinator				Service				Coordinator				Service to replicate				Redundant Infrastructure			
	Scenarios	T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4
Pool	Phone sales agent																												
	* GetKeyManager(CRMReq) : key_mng																												
	* CallClient(name_client, phone)																												
	Client																												
	Direct sales agent																												
	* OfferLoanContract																												
	* ReportSupervisor																												
	* SignVisitClientDoc																												
Services	S_EMAIL_MNG		P8	P8	P8										P14	P14	P14												
	* GetMsgDetails(IdQ) : date_msg, email_add						P8																						
	S_CRM		P9	P9	P9										P14	P14	P14												
	* CustName(email_address) : name_cust						P9																						
	* CustDetails(key_mng, name_cust) : address_cust, phone, email_cust		P10	P10	P10																								
	S_BANK_ACC						P10	P10	P10						P14	P14	P14							P15	P15				
	* ClientName(email_address) : name_client						P10	P10	P10																				
	* FundsAcc(ID_client, account_num) : funds_account						P10	P10	P10																				
	* IDClient(name_client) : ID_client						P10	P10	P10																				
	* AccNumber(ID_client) : account_num						P10	P10	P10																				
	S_SALES		P11	P11	P11										P14	P14	P14								P16				
	* CalculateOffer(delta_goal, funds_account) : offer_client						P11	P11	P11																				
	* SalesQuarter(Quarter) : salesQuarter						P11	P11	P11																				
	S_PLANNING		P12	P12	P12										P14	P14	P14												
	* CalculateDeltaGoal(goalQuarter, salesQuarter) : delta_Goal						P12	P12	P12																				
	S_LOAN-CLIENT		P13	P13	P13										P14	P14	P14												
	* OfferLoanClient (email_adress) : offer_client						P13	P13	P13																				
Application Components	EMAIL_MNG																												
	* MsgDetails(IdQ) : date_msg, email_add																												
	CRM																												
	* Get_CustName(email_address) : name_cust																												
	* Get_CustDetails(key_mng, name_cust) : address_cust, phone, email_cust																												
	BANK_ACC																												
	* Get_ClientName(email_address) : name_client																												
	* Get_FundsAcc(ID_client, account_num) : funds_account																												
	* Get_IDClient(name_client) : ID_client																												
	* Get_AccNumber(ID_client) : account_num																												
	SALES																												
	* CalculateOffer(delta_goal, funds_account) : offer_client																												
	* Get_SalesQuarter(Quarter) : salesQuarter																												
	PLANNING																												
	* CalculateDeltaGoal(goalQuarter, salesQuarter) : delta_Goal																												
	REDU_MNG																			P15									
	CMM										P14	P14																	
	ESB												P14								P15, P16								
Infrastructure	N1																											P15	P15
	N2																											P15	P15
	N3																											P15	P!5
	N4																											P15	P16
	N5																											P15	P16

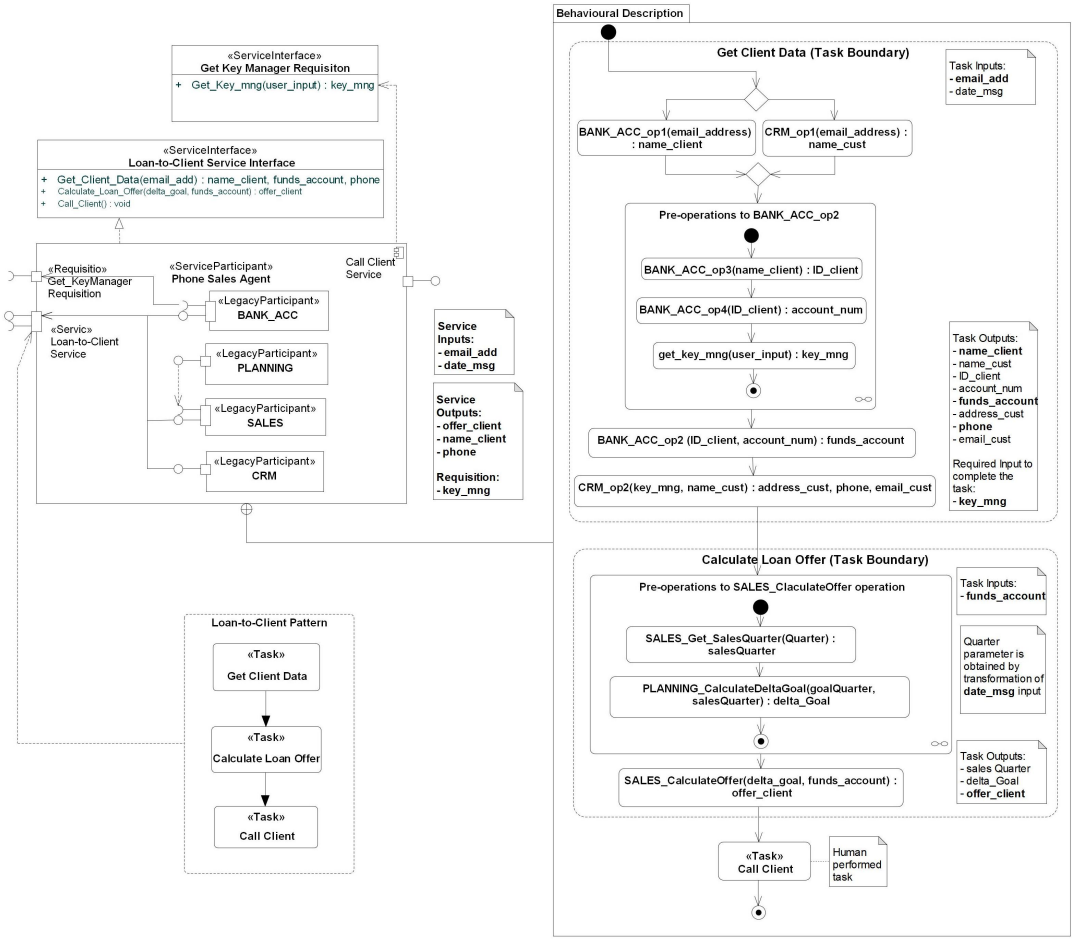


Figure C.18: *Loan to Client* pattern traced to its associated service and implementation.

Appendix D

Main Source Code for Algorithms

Contents

D.1 Matlab Code for Structural Matching and Discovery

This section provides the matlab code for functions used in matching and discovery algorithms from Chapters 5 and 6. Functions names, inputs, outputs and dependencies are described first. Afterwards, the Matlab code are provided in the same order functions were exposed. Figure D.1 indicates the function dependencies.

MatchPattern

Function MatchScore = MatchPattern(GraphMatrix, GraphLabels, PatternMatrix, PatternLabels)

Input:
Graph Matrix (adjacency matrix of undirected graph model, size = m x m)
Graph Label (labels of vertices and edges from graph model, column vector, length = m)
Pattern Matrix (adjacency matrix of undirected graph pattern, size = n x n)
Pattern Label (labels of vertices and edges of graph pattern, column vector, length = n)

Output:
Indexed matches for Graph Label (column vector, length = m)

Depdendencies:
GetAmebaElementList
GetAmebaNeighbors3
MatchVertex

$*$ Observation: The simplest case for `MatchVertex` function is consider the `strcmp` function to compare strings describing two vertices. Types and attributes comparison between vertices are discussed in Chapter 5.5.

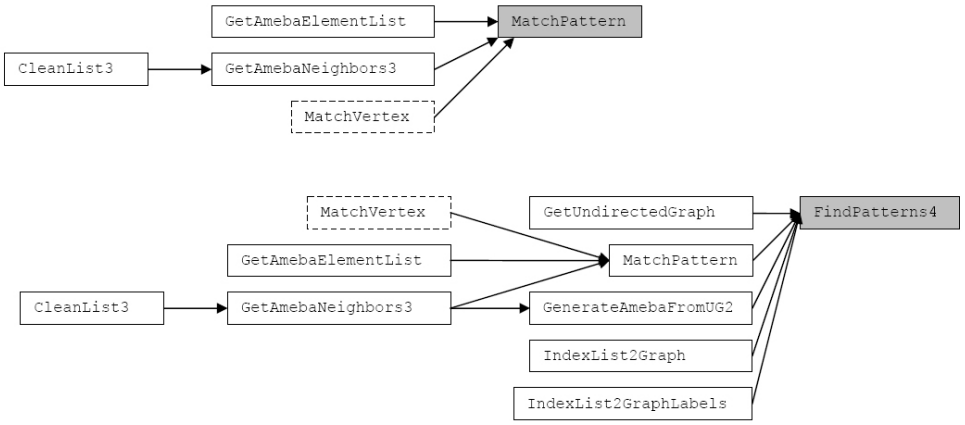


Figure D.1: Function dependencies for matching and discovery (find) algorithms.

GetAmebaElementList

Function Amembers=GetAmebaElementList(amebaIndex, AmebaList)

Input:
amebaIndex (number in AmebaList)
AmebaList (row-vector of numbers with arbitrary length, numbers can represent positions of vertices in a graph label vector)

Output:
Amembers

Dependencies:
none

cleanList3

Function CleanLista=cleanList3(Lista)

Input:
Lista (column-vector, e.g. [1; 2; 3; 3; 4; 4; 4])

Output:
CleanLista (clean column-vector without repeated elements, answer from example above [1; 2; 3; 4])

Dependencies:
none

FindPatterns4

Function [ScoreFoundPatterns, FreqMatrix]=FindPatterns4(GraphMatrix, GraphLabels, stepsPattern,

Threshold)

Input:
GraphMatrix
GraphLabels
stepsPattern
Threshold (approximate match ratio Rt)

Output:
ScoreFoundPatterns
FreqMatrix

Dependencies:
GetUndirectedGraph
MatchPattern
GenerateAmebaFromUG2
IndexList2Graph
IndexList2GraphLabels

GetUndirectedGraph

Function [uGraph, uLabelsGraph]=GetUndirectedGraph(graph, labelsGraph)

Input:
Graph
labelsGraph

Output:
uGraph
uLabelsGraph

Dependencies:
none

GenerateAmebaFromUG2

Function AmebaIndexList=GenerateAmebaFromUG2(Graph,GraphLabels,iAmebaIndexList,steps)

Input:
Graph
GraphLabels
iAmebaIndexList
steps

Output:
AmebaIndexList

Dependencies:
GetAmebaNeighbors3

IndexList2Graph

Function GraphFromIndexList=IndexList2Graph(IndexList,Graph)

input:
IndexList
Graph

output:
GraphFromIndexList

depdendencies:
none

IndexList2GraphLabels

Function GraphLabelsFromIndexList=IndexList2GraphLabels(IndexList,GraphLabels)

input:
IndexList
GraphLabels

output:
GraphLabelsFromIndexList

depdendencies:
none

```
function nMatchScore=MatchPattern(GraphMatrix, GraphLabels,
                    PatternMatrix, PatternLabels)

tic;
MatchScore = zeros(size(GraphLabels));

for indexP=1:length(PatternLabels)
    for indexG=1:length(GraphLabels)
        if MatchVertex(PatternLabels(indexP),GraphLabels(indexG))==1 %*Observation
            MatchScore(indexG)=1;
        end
    end
end

AmebaList=zeros(size(MatchScore));
for i=1:length(MatchScore)
    if MatchScore(i)==1
        AmebaList(i)=sum(MatchScore(1:i));
        LastAmebaIndex=AmebaList(i);
    end
end


for n=1:LastAmebaIndex
    amebaIndex=n;
```

```
    flag=1;
    while flag==1
        change=0;
        Amembers=GetAmebaElementList(amebaIndex, AmebaList);
        vacia=isempty(Amembers);
        if vacia≠1
            Aneighbors=GetAmebaNeighbors3(GraphMatrix,Amembers);
            for j=1:length(AmebaList)
                if (AmebaList(j)≠0) && (AmebaList(j)≠amebaIndex)
                    for k=1:size(Aneighbors,1)
                        if j==Aneighbors(k)
                            AmebaList(j)=amebaIndex;
                            change=1;
                        end
                    end
                end
            end
        end
        if (change==0)
            flag=0;
        end
    end
end


AmebaWeights=zeros(LastAmebaIndex,1);

for n=1:LastAmebaIndex
    for j=1:length(AmebaList)
        if AmebaList(j)==n
            AmebaWeights(n)=AmebaWeights(n)+1;
        end
    end
end


MatchScore=AmebaList;
nMatchScore=AmebaList;

for n=1:LastAmebaIndex
    if AmebaWeights(n)≠0
        for j=1:length(MatchScore)
            if MatchScore(j)==n
                nMatchScore(j)=AmebaWeights(n);
            end
        end
    end
end

time=toc
```

```
function Amembers=GetAmebaElementList(amebaIndex, AmebaList)

Amembers=0;

for i=1:length(AmebaList)
    if AmebaList(i)== amebaIndex
        Amembers=[Amembers; i];
    end
end

Amembers=Amembers(2:length(Amembers));
```

```
function CleanLista=cleanList3(Lista)

sz=size(Lista(:,1));
CleanLista=0;

for n=1:sz
   flag=0;
```

```
    for m=1:length(CleanLista)
     if (CleanLista(m)==Lista(n,1)) flag=1; end
    end
    if (flag==0) CleanLista=[CleanLista; Lista(n,1)]; end
end

CleanLista=CleanLista(2:size(CleanLista(:,1)));
```

```
function [ScoreFoundPatterns, FreqMatrix]=FindPatterns4(GraphMatrix, GraphLabels, stepsPattern, Threshold)

[uGraphMatrix, uGraphLabels]=GetUndirectedGraph(GraphMatrix, GraphLabels);

for i=1:size(uGraphMatrix,1)

    AmebaIndexList(i,1)={[i]};
    tmp2=IndexList2Graph(AmebaIndexList{i,1}',uGraphMatrix);
    uPatternMatrix(i,1)={tmp2};
    tmp3=IndexList2GraphLabels(AmebaIndexList{i,1}',uGraphLabels);
    uPatternLabels(i,1)={tmp3};
    tmp=MatchPattern(uGraphMatrix, uGraphLabels, uPatternMatrix{i,1}, uPatternLabels{i,1});
    ScoreFoundPatterns(i,1)={tmp};
    aux1=ScoreFoundPatterns{i,1}./length(uPatternLabels{i,1});

    cnt1=0;
    for k=1:length(aux1)
        if aux1(k)≥Threshold
            cnt1=cnt1+1;
        end
    end

    estimateFreq1=cnt1/length(uPatternLabels{i,1});
    FreqMatrix(i,1)=estimateFreq1;

    if stepsPattern>0
        for j=1:stepsPattern
            tmp4=GenerateAmebaFromUG2(uGraphMatrix,uGraphLabels,AmebaIndexList{i,j}',1);
            AmebaIndexList(i,j+1)={tmp4};
            tmp2=IndexList2Graph(AmebaIndexList{i,j+1}',uGraphMatrix);
            uPatternMatrix(i,j)={tmp2};
            tmp3=IndexList2GraphLabels(AmebaIndexList{i,j+1}',uGraphLabels);
            uPatternLabels(i,j)={tmp3};
            tmp=MatchPattern(uGraphMatrix, uGraphLabels, uPatternMatrix{i,j}, uPatternLabels{i,j});
            ScoreFoundPatterns(i,j+1)={tmp};
            aux2=ScoreFoundPatterns{i,j+1}./length(uPatternLabels{i,j});

            cnt2=0;
            for m=1:length(aux2)
                if aux2(m)≥Threshold
                    cnt2=cnt2+1;
                end
            end

            estimateFreq2=cnt2/length(uPatternLabels{i,j});
            FreqMatrix(i,j+1)=estimateFreq2;
        end
    end

end
```

```

function [uGraph, uLabelsGraph]=GetUndirectedGraph(graph, labelsGraph)

uLabelsGraph={['0']};
for i=1:length(labelsGraph)
   uLabelsGraph=[uLabelsGraph; labelsGraph(i)];
end


for i=1:length(labelsGraph)
    for j=1:length(labelsGraph)
```

```
        uGraph(i,j)=0;
    end
end

count=0;
for i=1:length(labelsGraph)
    for j=1:length(labelsGraph)
      if graph(i,j)==1
        count=count+1;
        uLabelsGraph=[uLabelsGraph; strcat(labelsGraph(i),'_',labelsGraph(j))];
           % added: '_' to separate numeric labels
        uGraph(i,(length(labelsGraph)+count))=1;
        uGraph(j,(length(labelsGraph)+count))=1;
        uGraph((length(labelsGraph)+count),i)=1;
        uGraph((length(labelsGraph)+count),j)=1;
        uGraph((length(labelsGraph)+count),(length(labelsGraph)+count))=0;
      end
    end
end

uLabelsGraph=uLabelsGraph(2:length(uLabelsGraph));
```

```

function AmebaIndexList=GenerateAmebaFromUG2(Graph,GraphLabels,iAmebaIndexList,steps)

AmebaIndexList=iAmebaIndexList;
AmebaNeighborList=GetAmebaNeighbors3(Graph,AmebaIndexList);

for i=1:steps %Ameba grow 'n' steps
    for j=1:size(AmebaNeighborList,1)
        AmebaIndexList=[AmebaIndexList AmebaNeighborList(j,1)];
    end
    AmebaNeighborList=GetAmebaNeighbors3(Graph,AmebaIndexList);
end

AmebaIndexList=AmebaIndexList';
AmebaNeighborList;
```

```

function GraphFromIndexList=IndexList2Graph(IndexList,Graph)

for i=1:size(IndexList,2)
    for j=1:size(IndexList,2)
        GraphFromIndexList(i,j)=Graph(IndexList(1,i),IndexList(1,j));
    end
end
```

```

function GraphLabelsFromIndexList=IndexList2GraphLabels(IndexList,GraphLabels)

for i=1:size(IndexList,2)
    GraphLabelsFromIndexList(i)=GraphLabels(IndexList(i));
end

GraphLabelsFromIndexList=GraphLabelsFromIndexList';
```

D.2 Matlab Code Experiments and Visualisation Functions

This section describes the matlab code for complementary functions used in Chapter 8. Functions names, inputs, outputs and dependencies are described first. Afterwards, the Matlab code are provided in the same order functions were exposed.

genPosSample

Function [sizePattern,PatternIndexList,TESTAdjP,TESTLabP,uTESTAdjP,uTESTLabP]=
genPosSample(AdjM,LabM,index,steps)
Input:
AdjM: Adjacency model graph matrix (directed graph).
LabM: Vector with labels for model graph vertices (directed graph).
index: index of vertex from where the positive pattern sample is generated after expansion steps with neighbours.
steps: number of expansion steps.

Output:
sizePattern: number of vertices in the pattern sample.
PatternIndexList: List with indexes of vertices in pattern sample. TESTAdjP: Adjacency pattern graph matrix (directed graph). TESTLabP: Vector with labels for pattern graph vertices (directed graph). uTESTAdjP: Adjacency pattern graph matrix (undirected graph). uTESTLabP: Vector with labels for pattern graph vertices (undirected graph).
Dependencies:
GenerateAmebaFromUG2
GetUndirectedGraph

genNegSample

Function [AdjSum,LabSum,antiPAdj,antiPLab,InvantiPAdj,rowInvantiPLab,colInvantiPLab,NegPAdj, NegPLabRow,NegPLabCol]=genNegSample(Adj,Lab,szP,index)
Input:
Adj: Adjacency model graph matrix (directed graph).
Lab: Vector with labels for model graph vertices (directed graph).
szP: number of vertices in pattern graph.
index: index in matrix without repeated labels (derived from adjacency matrix) where the adjacency matrix of the negative sample pattern is derived from.
Output:
AdjSum: Adjacency matrix without repeated labels derived from input adjacency matrix (Adj).
LabSum: Vector with only different labels derived from input vector with labels (Lab).
InvantiPAdj: Preparatory negative sample pattern matrix from where is extracted NegPAdj (it is not an adjacency matrix). NegPAdj: Negative sample pattern matrix (it is not an adjacency matrix). NegPLabRow: Vector with row labels for pattern graph. NegPLabCol: Vector with column labels for pattern graph.
Dependencies:
none

testSample1

Function testSample1(AdjM, LabM, AdjP, LabP, uAdjM, uLabM, uAdjP, uLabP, Type, Sample)
Input:
AdjM: Adjacency model graph matrix (directed graph).
LabM: Vector with labels for model graph vertices (directed graph).
AdjP: Adjacency pattern graph matrix (directed graph).
LabP: Vector with labels for pattern graph vertices (directed graph).
uAdjM: Adjacency model graph matrix (undirected graph).
uLabM: Vector with labels for model graph vertices (undirected graph).
uAdjP: Adjacency pattern graph matrix (undirected graph).
uLabP: Vector with labels for pattern graph vertices (undirected graph).
TYPE: string 'POS' or 'NEG' for positive and negative samples.
SAMPLE: string indicating the sample number.

Output:
Creates .DOT files with directed and undirected versions of the target graphs and results of the matching process. The files can be opened in GraphViz and the pattern matching results visualised. Matched vertices and edges are coloured in green.
Dependencies:
MatchPattern
IndexList2GraphLabels
GetSourceTargetFromEdge
graph_to_dot5

testDiscovery1

Function [ScoreFoundPatterns, FreqMatrix,ARCLABEL]=testDiscovery1(GraphMatrix, GraphLabels, varargin)
Input:
GraphMatrix: Adjacency model graph matrix (directed graph).
GraphLabels: Vector with labels for model graph vertices (directed graph).
varargin: arguments for variables stepsPattern, Threshold, FreqInterest and NameModel. See comments on source code.
Output:
ScoreFoundPatterns: Score found patterns matrix (*SFPM*). See Section 8.6 for details. FreqMatrix: Frequency matrix (*FM*). See Section 8.6 for details. ARCLABEL: Matrix indicating the arcs between vertices of the discovered patterns. It is used for to create the .DOT files for visualisation purposes. testDiscovery1 also creates .DOT files with the undirected version of the target graph and the results of the discovery process. The files can be opened in GraphViz and the pattern discovery results visualised (both files have coloured vertices, only one with coloured edges).
Dependencies:
GetUndirectedGraph
FindPatterns4
GenerateAmebaFromUG2
IndexList2GraphLabels
GetSourceTargetFromEdge
graph_to_dot5

rndExp1

Function [Match,uMatch]=rndExp1(szM,LszM,szP,LszP)

The function generates a random graph model and a random graph pattern, then it performs the pattern matching function with the randomly generated pattern and model. Later, it writes the matches on the graph model as a .dot file that can be visualised with graphviz.

Input:
szM: integer representing the number of vertices of the graph model.
LszM: integer representing the number of different labels for graph model vertices. A label in this case is a "number" converted to string.
szP: integer representing the number of vertices of the graph pattern.
LszP: integer representing the number of different labels for graph pattern vertices. A label in this case is a "number" converted to string.

Output:
Match: match result (vector associated to each vertex's index) with the amount of matched vertices indicated with an integer. The matching is for the "directed" version of the graph. Example: if 3 vertices belong to a match, and these vertices are 4, 23 and 15, then Match(4)=3, Match(23)=3 and Match(15)=3.
uMatch: match result (vector associated to each vertex's index) with the amount of matched vertices indicated with an integer. The matching is for the 'UNdirected' version of the graph. Example: if 2 vertices belong to a match, and these vertices are 4, 5 (and 4-5) the match would preserves the direction of edges, and Match(4)=3, Match(4-5)=3 and Match(5)=3.

Dependencies:
matrand2
matrand2Pattern
MatchPattern
graph_to_dot5
GetUndirectedGraph

matrand2

Function [MR,LMR]=matrand2(sz2,Lsz)

The function generates a random "process" graph. This type of graph contains a source and sink vertices and all vertices have a bounded in/out-degree. The function does not ensure dead-lock free graphs and in some rare cases it mights generate disconnected graphs. This type of graph can be discarded by other functions processing the graph.

Input:
sz2: integer representing the number of vertices of the graph model.
Lsz: integer representing the number of different labels for graph model vertices.

Output:
MR: A matrix representing the connectivity between vertices of the random "process" graph.
LMR: A vector containing labels for vertices of the random "process" graph. Labels can be alternatively generated for different external functions. In this case it is illustrated with internal steps generating random numbers between 1 and Lsz, which are then converted to strings to represent labels.

rndMatchModel

Function [Match,uMatch]=rndMatchModel(szM,LszM,adjP,LP)

The function generates a random graph model and uses a given pattern with adjacency matrix = adjP and label vector = LP. Later a pattern matching step is performed and finally the function writes the resultant matches over the graph model. They can be visualised using GraphViz. LszM allows labelling with numbers between 1 and LszM. LP must refer to numbers.

Input:
szM: integer representing the number of vertices of the graph model
LszM: integer representing the number of different labels for graph model vertices. A label in this case is a "number" converted to string.
adjP: adjacency matrix of the graph pattern
LP: labels vector of the graph pattern. Each label is a "number" converted to string.

Output:
Match: match result (vector associated to each vertex's index) with the amount of matched vertices indicated with an integer. The matching is for the 'directed' version of the graph. Example: if 3 vertices belong to a match, and these vertices are 4, 23 and 15, then Match(4)=3, Match(23)=3 and Match(15)=3.
uMatch: match result (vector associated to each vertex's index) with the amount of matched vertices indicated with an integer. The matching is for the 'UNdirected' version of the graph. Example: if 2 vertices belong to a match, and these vertices are 4, 5 (and 4-5) the match would preserves the direction of edges, and Match(4)=3, Match(4-5)=3 and Match(5)=3.

Dependencies:
matrand2
MatchPattern
graph_to_dot5
GetUndirectedGraph
(use PatternSamples1 : only if using predefined patterns in this file)

matches-to-dot

Function matches-to-dot(M,LM,P,LP)

Initially this function performs the pattern matching function between M and P. Later, it generates .dot files for M (M.dot), P (P.dot) and the matched M (Matches.dot). The same is done for the undirected version of M and P (uM and uP). The matched model is represented in the uMatches.dot file. All .dots files can be visualised with GraphViz.

Output (no in Matlab enviroment): M.dot, uM.dot, P.dot, uP.dot, Matches.dot, uMatches.dot

∗ Observation: Copyrights to authors in Matlab Code `graph\_to\_dot5`.

graph_to_dot5

Modified version of graph-to-dot (2004) by Dr. Leon Peshkin, pesha @ ai.mit.edu / pesha.
See Matlab Code in listings at the end of the section.

GraMat_to_GraML

Function GraMat_to_GraML(adj, attrName, attrType, attr, varargin)

Input:
adj: graph adjacency matrix
attrName: vector with n entries representing the names of attributes for each graph vertex.
attrType: vector with n entries representing the types of attributes for each graph vertex.
attr: cell matrix with attribute values for each vertex. Rows represent the vertices of the graph, columns represent the attribute values.
varargin: used to change the default name of the output file (GraphML.xml).

Output:
A GraphML file (XML-based format). More details in http://graphml.graphdrawing.org/.

```
1
2  function [sizePattern,PatternIndexList,TESTAdjP,TESTLabP,uTESTAdjP,uTESTLabP]=genPosSample(AdjM,
3  LabM,index,steps)
4
5  %generate a positive pattern sample
6  PatternIndexList=GenerateAmebaFromUG2(AdjM,LabM,index,steps);
7  %PatternIndexList is the index list in adjacency matrix 'AdjM'
8
9  sizePattern = length(PatternIndexList);
10
11 for i=1:sizePattern
12     Ind = PatternIndexList(i);
13     LabP(i)=LabM(Ind);
14 end
15
16 for i=1:sizePattern
17     for j=1:sizePattern
18         IndI = PatternIndexList(i);
19         IndJ = PatternIndexList(j);
20         AdjP(i,j)=AdjM(IndI,IndJ);
21     end
22 end
23
24 TESTAdjP=AdjP;
25 TESTLabP=LabP';
26 [uAdjP,uLabP]=GetUndirectedGraph(AdjP, LabP);
27 uTESTAdjP=uAdjP;
28 uTESTLabP=uLabP;
```

```
1
2  function [AdjSum,LabSum,NegPAdj,NegPLabRow,NegPLabCol]=genNegSample(Adj,Lab,szP,index)
```

```
3
4  %use this to review all temporal outputs:
5  %[AdjSum,LabSum,antiPAdj,antiPLab,InvantiPAdj,rowInvantiPLab,colInvantiPLab,NegPAdj,
6  %NegPLabRow,NegPLabCol]
7
8  SumRowAdj=Adj;
9  SumRowLab=Lab;
10 flagRow = zeros(length(Lab),1);
11
12
13 for i=1:length(Lab) % recorre columns
14     for j=1:length(Lab) % recorre rows
15         if (i≠j) && (strcmp(Lab(j),Lab(i))==1) && (flagRow(i)==0)
16             SumRowAdj(i,:)=Adj(j,:)+Adj(i,:); % add rows i and j
17             flagRow(i)=1; % manterner sumada
18             flagRow(j)=2; % eliminar
19         end
20     end
21 end
22
23 TflagRow=flagRow;
24 cnt=0;
25 for k=1:length(TflagRow)
26         if flagRow(k)≠2
27             cnt=cnt+1;
28         end
29 end
30
31 nSumRowAdj=zeros(cnt,length(Lab));
32 nSumRowLab=cell(cnt,1);
33 for k=1:length(TflagRow)
34     if flagRow(k)≠2
35         %k;
36         nSumRowAdj(k,:)=SumRowAdj(k,:);
37         nSumRowLab(k)=SumRowLab(k);
38     end
39 end
40
41 SumColAdj=nSumRowAdj;
42 SumColLab=nSumRowLab;
43 flagCol = zeros(length(SumColLab),1);
44 k2=size(nSumRowAdj,1); % number of rows
45 k3=size(nSumRowAdj',1); % number of columns
46
47 for i=1:k2 % recorre rows
48     for j=1:k3 % recorre columns
49         if (j≠i) && (strcmp(nSumRowLab(i),Lab(j))==1) && (flagCol(i)==0)
50             %j;
51             %i;
52             SumColAdj(:,i)=nSumRowAdj(:,i)+nSumRowAdj(:,j);
53             % add columns i and j
54             flagCol(i)=1; %manterner sumada
55             flagCol(j)=2; % eliminar
56         end
57     end
58 end
59
60 TflagCol=flagCol;
61 cnt2=0;
62 for m=1:length(TflagCol)
63         if flagCol(m)≠2
64             cnt2=cnt2+1;
65         end
66 end
67
68 nSumColAdj=zeros(length(nSumRowLab),cnt2);
69 nSumColLab=cell(cnt2,1);
70 for m=1:length(TflagCol)
71     if flagCol(m)≠2
72         %m;
73         nSumColAdj(:,m)=SumColAdj(:,m);
74         nSumColLab(m)=SumColLab(m);
75     end
76 end
77
```

```
AdjSum=nSumColAdj;
LabSum=nSumColLab;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%index;
j2=size(nSumColAdj,1); %number of rows matrix without repeated labels
antiPAdj=zeros(szP,j2);
antiPLab=cell(szP,1);
if (index > j2-szP)==1
    %ee = j2-szP+1;
    e = 'error: invalid index (index must be ≤ (Adj rows - szP),
    change to default index = 1)'
    index=1;
end
j3=index+szP;
for i=1:szP
    antiPAdj(i,:)=nSumColAdj(index+i,:);
    antiPLab(i)=nSumColLab(index+i);
end

l2=size(antiPAdj,1); %number of rows
l3=size(antiPAdj',1); %number of columns
InvantiPAdj=zeros(l2,l3);
for i=1:l2
    for j=1:l3
        if antiPAdj(i,j)==1
            InvantiPAdj(i,j)=0;
        end
        if antiPAdj(i,j)==0
            InvantiPAdj(i,j)=1;
        end
    end
end

rowInvantiPLab=antiPLab; % labels in rows
colInvantiPLab=SumColLab; % lables in columns

NegPAdj=InvantiPAdj(1:szP,1:szP);
NegPLabRow=rowInvantiPLab;
NegPLabCol=colInvantiPLab(1:szP);
```

```

function testSample1(AdjM, LabM, AdjP, LabP, uAdjM, uLabM, uAdjP, uLabP, Type, Sample)

%TYPE = POS or NEG, for positive and negative samples, negative sample
%indicate a pattern that does not exist in the graph.
%Write 'POS' or 'NEG' as string

%SAMPLE indicate the number of the sample, write '1' as a string to
% indicate it is the first sample, '2', the second sample, etc.

%the matching
Match=MatchPattern(AdjM, LabM, AdjP, LabP);
uMatch=MatchPattern(uAdjM, uLabM, uAdjP, uLabP);

%steps to add labels and weights in each vertex of the 'match'
uS_Match=cell(length(uMatch),1);
S_Match=cell(length(Match),1);

uMatch_labels=cell(length(uMatch),1);
Match_labels=cell(length(Match),1);

for i=1:length(uMatch)
    uS_Match(i)={num2str(uMatch(i))};
    S_i={num2str(i)};
    uMatch_labels(i)=strcat(uLabM(i),' [',uS_Match(i),']',',i=',S_i);
end
for i=1:length(Match)
    S_Match(i)={num2str(Match(i))};
    S_i={num2str(i)};
    Match_labels(i)=strcat(LabM(i),' [',S_Match(i),']',',i=',S_i);
```

```
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% write to "Directed" and "Undirected" graphs without colouring edges

UdotFileName = strcat('uRESULT-M',num2str(length(LabM)),'-P',num2str(length(LabP)),'-',
Type,'-S',Sample,'.dot');

dotFileName = strcat('RESULT-M',num2str(length(LabM)),'-P',num2str(length(LabP)),'-',
Type,'-S',Sample,'.dot');

graph_to_dot5(uAdjM,'filename',UdotFileName,'node_label',uMatch_labels,'matches',uMatch);
graph_to_dot5(AdjM,'filename',dotFileName,'node_label',Match_labels,'matches',Match);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% write to "Directed" graph colouring edges

% uGraphLabels is vector with undirected graph labels
% from 1 to size of directed graph [i.e length(GraphLabels)], labels are the same
% as directed graph labels.

Edges=length(uLabM)-length(LabM);
Edge_Labels=cell(Edges,1);
SourceLabel=cell(Edges,1);
TargetLabel=cell(Edges,1);

arc_label=zeros(length(LabM)); % for graph_to_dot4 -> write color to edge in directed graph

for i=1:length(LabM)
    Edge_Labels(i,1)={'0'};
    SourceLabel(i,1)={'0'};
    TargetLabel(i,1)={'0'};
end

for i=length(LabM)+1:length(uLabM)
    if uMatch(i)>0
        Edge_Labels(i,1)=IndexList2GraphLabels(i,uLabM);
        indexEdge=i;
        [IndexSource,IndexTarget]=GetSourceTargetFromEdge(indexEdge,uLabM,uAdjM);
        arc_label(IndexSource,IndexTarget)=1;
        STLabels=textscan(Edge_Labels{i,1},'%[^_]%1c%s');
        SourceLabel{i,1}=STLabels{1,1};
        TargetLabel{i,1}=STLabels{1,3};
    else
        Edge_Labels{i,1}='0';
        SourceLabel{i,1}='0';
        TargetLabel{i,1}='0';
    end
end

ARCLABEL=arc_label;

DirectMatch=uMatch(1:length(LabM));

dotFileName2 = strcat('MATCHING-D--M',num2str(length(LabM)),'-P',num2str(length(LabP)),'-',
Type,'-S',Sample,'.dot');

graph_to_dot5(AdjM,'filename',dotFileName2,'node_label',uLabM,'edgeColor',ARCLABEL,
'matches',DirectMatch);
```

```

function [ScoreFoundPatterns, FreqMatrix,ARCLABEL]=
testDiscovery1(GraphMatrix, GraphLabels, varargin)

% Default: stepsPattern, Threshold, FreqInterest, NameModel
stepsPattern = 1; % one expansion step
Threshold = 1; % exact vertex matching
FreqInterest = 2; % pattern at least two times in the model
NameModel = []; % no name

% Selection: stepsPattern, Threshold, FreqInterest, NameModel
for i = 1:3:nargin-1              % optional args
```

```
    switch varargin{i}
        case 'stepsPattern', stepsPattern = varargin{i+1};
        case 'Threshold', Threshold = varargin{i+1};
        case 'FreqInterest', Threshold = varargin{i+1};
        case 'NameModel', NameModel = varargin{i+1};
    end
end

%'NameModel' is a string indicating the name of the model
% identical to v7 test discovery with comments

[uGraphMatrix, uGraphLabels]=GetUndirectedGraph(GraphMatrix, GraphLabels);
%the discovery
[ScoreFoundPatterns, FreqMatrix]=FindPatterns4(GraphMatrix, GraphLabels,
stepsPattern, Threshold);

sp=stepsPattern+1;
MatchDisc1=zeros(length(FreqMatrix(:,sp)),1);

if stepsPattern≤1 % 1 step of expansion or less
   for i=1:length(FreqMatrix(:,sp))
    if FreqMatrix(i,sp)>1  % discovered pattern is at least 2 times in the model
        MatchDisc1(i)=FreqMatrix(i,sp);
    else
        MatchDisc1(i)=0;
    end
   end
else
    for i=1:length(FreqMatrix(:,sp)) % more than 1 step of expansion
     if FreqMatrix(i,sp)≥FreqInterest  % FreqInterest = 2 means
     % discovered pattern is atleast 2 times in the model
        MatchDisc1(i)=FreqMatrix(i,sp);
     else
        MatchDisc1(i)=0;
     end
    end
end

MatchDisc2=MatchDisc1;
for i=1:length(MatchDisc1)
    if MatchDisc1(i)≠0
        iAmebaIndexList=i;
        AmebaIndexList=GenerateAmebaFromUG2(uGraphMatrix,uGraphLabels,iAmebaIndexList,sp);
        for j=1:length(AmebaIndexList)
            indexNeigbour=AmebaIndexList(j);
            if MatchDisc1(indexNeigbour)≥2
                MatchDisc2(indexNeigbour)=MatchDisc1(indexNeigbour);
            else
                MatchDisc2(indexNeigbour)=1;
            end
        end
    end
end

MatchDisc3=MatchDisc2;
cnt=zeros(length(MatchDisc2),1);
for i=1:length(MatchDisc2)
    for j=1:length(MatchDisc2)
        if MatchDisc2(i)==1
            if MatchDisc2(j)==1
                LabelFromIndex_i=IndexList2GraphLabels(i,uGraphLabels);
                LabelFromIndex_j=IndexList2GraphLabels(j,uGraphLabels);
                if strcmp(LabelFromIndex_i,LabelFromIndex_j)==1  % verify if the
                % label is more than one time in Model
                    cnt(i)=cnt(i)+1;
                    cnt(j)=cnt(j)+1;
                end
            end
        end
    end
end

for i=1:length(cnt)
    if cnt(i)>2
        MatchDisc3(i)=1;
```

```
88      else
89          MatchDisc3(i)=0;
90      end
91  end
92
93  MatchDisc4=MatchDisc3+MatchDisc1;
94
95  S_stepsPattern=num2str(stepsPattern);
96
97  % write to "Undirected" graph
98  UdotFileName = strcat('DISCOVERY_U_',NameModel,'_M',
99  num2str(length(GraphLabels)),'_steps',S_stepsPattern,'.dot');
100
101 %graph_to_dot3(uGraphMatrix,'filename',UdotFileName,'node_label',
102 uGraphLabels,'matches',MatchDisc4);
103
104 graph_to_dot5(uGraphMatrix,'filename',UdotFileName,'node_label',
105 uGraphLabels,'matches',MatchDisc4);
106
107
108 % write to "Directed" graph
109
110 % uGraphLabels is vector with undirected graph labels
111 % from 1 to size of directed graph [i.e length(GraphLabels)],
112 % labels are the same as directed graph labels.
113
114 Edges=length(uGraphLabels)-length(GraphLabels);
115 Edge_Labels=cell(Edges,1);
116 SourceLabel=cell(Edges,1);
117 TargetLabel=cell(Edges,1);
118
119 arc_label=zeros(length(GraphLabels));
120
121 for i=1:length(GraphLabels)
122     Edge_Labels(i,1)={'0'};
123     SourceLabel(i,1)={'0'};
124     TargetLabel(i,1)={'0'};
125 end
126
127 for i=length(GraphLabels)+1:length(uGraphLabels)
128     if MatchDisc4(i)>0
129         Edge_Labels(i,1)=IndexList2GraphLabels(i,uGraphLabels);
130         indexEdge=i;
131         [IndexSource,IndexTarget]=GetSourceTargetFromEdge(indexEdge,
132         uGraphLabels,uGraphMatrix);
133         arc_label(IndexSource,IndexTarget)=1;
134         STLabels=textscan(Edge_Labels{i,1},'%[^_]%1c%s');
135         SourceLabel{i,1}=STLabels{1,1};
136         TargetLabel{i,1}=STLabels{1,3};
137     else
138         Edge_Labels{i,1}='0';
139         SourceLabel{i,1}='0';
140         TargetLabel{i,1}='0';
141     end
142 end
143
144 ARCLABEL=arc_label;
145
146 DirectMatchDisc=MatchDisc4(1:length(GraphLabels));
147
148 dotFileName = strcat('DISCOVERY_D_',NameModel,'_M',num2str(length(GraphLabels)),
149 '_steps',S_stepsPattern,'.dot');
150
151 graph_to_dot5(GraphMatrix,'filename',dotFileName,'node_label',uGraphLabels,
152 'edgeColor',ARCLABEL,'matches',DirectMatchDisc);
```

```
1
2   function  [IndexSource,IndexTarget]=GetSourceTargetFromEdge(indexEdge,LabelsGraph,Graph)
3
4   at = find(LabelsGraph{indexEdge} == '_');
5       if at≠0
6           AmebaIndexList=GenerateAmebaFromUG2(Graph,LabelsGraph,indexEdge,1);
```

```
        if length(AmebaIndexList)==2 %self-loop
            IndexSource=AmebaIndexList(2,1);
            IndexTarget=AmebaIndexList(2,1);
        else
            s=textscan(LabelsGraph{indexEdge},'%[^_]%lc%s');
            if strcmp(LabelsGraph(AmebaIndexList(2,1)),s{1,1})==1
                IndexSource=AmebaIndexList(2,1);
                IndexTarget=AmebaIndexList(3,1);
            else
                IndexSource=AmebaIndexList(3,1);
                IndexTarget=AmebaIndexList(2,1);
            end
        end
    end
```

```

function Index=IndexFromLabel(Label,GraphLabels)

% provide Index vector indicating index of the vertex in
% GraphLabels that equals the 'Label' Index=zeros(length(GraphLabels),1);

for i=1:length(GraphLabels)
    if strcmp(Label,GraphLabels(i))==1
        Index(i)=i;
    end
end
```

```

function [Match,uMatch]=rndExp1(szM,LszM,szP,LszP)
%szM : size of the graph model
%szP : size of the graph pattern

% Better name: RndMatchModelPattern

%Generates random graph model and random graph pattern
%then it does the pattern matching step
% and writes the matches on the graph model

%added LszM and LszP 01 Dic. (it allows labelling with numbers between 1 and LszX, X=P,M)

[MR,LMR]=matrand2(szM,LszM);
[PR,LPR]=matrand2Pattern(szP,LszP);

%directed
Match=MatchPattern(MR, LMR, PR, LPR);
%add label with 'label' and weight of the 'match'
S_Match=cell(length(Match),1);
Match_labels=cell(length(Match),1);
for i=1:length(Match)
    S_Match(i)={num2str(Match(i))};
    Match_labels(i)=strcat(LMR(i),'(',S_Match(i),')');
end

graph\_to\_dot5(MR,'filename','MR.dot','node_label',LMR);
graph\_to\_dot5(PR,'filename','PR.dot','node_label',LPR);
graph\_to\_dot5(MR,'filename','MatchResult.dot','node_label',Match_labels,'matches',Match);

%undirected
[uMR, uLMR]=GetUndirectedGraph(MR, LMR);
[uPR, uLPR]=GetUndirectedGraph(PR, LPR);
uMatch=MatchPattern(uMR, uLMR, uPR, uLPR);
%add label with 'label' and weight of the 'match'
uS_Match=cell(length(uMatch),1);
uMatch_labels=cell(length(uMatch),1);
for i=1:length(uMatch)
    uS_Match(i)={num2str(uMatch(i))};
    uMatch_labels(i)=strcat(uLMR(i),'(',uS_Match(i),')');
end
graph\_to\_dot5(uMR,'filename','uMR.dot','node_label',uLMR);
graph\_to\_dot5(uPR,'filename','uPR.dot','node_label',uLPR);
```

```
graph\_to\_dot5(uMR,'filename','uMatchResult.dot','node_label',uMatch_labels,'matches',uMatch);
```

```

function [MR,LMR]=matrand2(sz2,Lsz) % equivalent to matrand2Pattern

% this function generates random "process" graphs. This type of graphs contain source and sink vertices
% and vertices have a bounded in/out-degree. The function does not ensure dead-lock free graphs and in
% some rare cases it mights generate disconnected graphs

sz=sz2-2; %graph size, less source and sink vertices
MRtemp=rand(sz);
MR=zeros(sz);

LMRtemp=rand(sz,1);
LMR=cell(sz,1);

sumColumn=zeros(sz,1)';
sumRow=zeros(sz,1)';

tmp1=0;
tmp2=0;

for j=1:sz
    for i=1:sz
        tmp1 = sumColumn(j);
        tmp2 = sumRow(i);
        tmp = tmp1 + tmp2;

        p1 = 1-1/sz;
        p2 = 1-(exp(-0.05*sz)); % (p1, p2, ...) are different functions to create edges.
        % p1 is used in this case. An edge is created with probability 1-1/sz
        % in small graphs vertices have more changes to connect to other vertices
        % in large graphs connections are less likely, and the in-out/degree results balanced
        % for different graph sizes

        if MRtemp(i,j)≥p1
            if tmp≤3      % max in/out-degree = 4, if fixed probability is considered (e.g 0.5 instead p1)
                MR(i,j)=1;
            else
                MR(i,j)=0;
            end
            sumColumn = sum(MR);
            sumRow = sum(MR');
        end

    end
end

% add row and column for source
t1=length(MR)+1;
MR(t1,:)=zeros(1,length(MR));
MR(:,length(MR))=zeros(length(MR),1);
iSource=t1;
%add row and column for sink
t2=length(MR)+1;
MR(t2,:)=zeros(1,length(MR));
MR(:,length(MR))=zeros(length(MR),1);
iSink=t2;
% add edges from source to all nodes without ingoing edges
for i=1:t2 % t2 = length(MR), goes through columns
    sumColumn_i = sum(MR(:,i));
    if (sumColumn_i==0) || ((sumColumn_i==1)&&(MR(i,i)==1)); % all nodes without ingoing edges OR
                                                             % without ingoing edges except
                                                             % selfloop
        MR(t1,i)=1;
        LMR(t1)={'Source'}; % uses SourcePattern for matrand2Pattern
    end
end
% add edges from all nodes without outgoing edges to the sink
for i=1:t2 % t2 = length(MR), goes through rows
    sumRow_i = sum(MR(i,:));
    if (sumRow_i==0) || ((sumRow_i==1)&&(MR(i,i)==1)); % all nodes without outgoing edges OR
```

```
                                                    % without outgoing edges except
                                                    % selfloop
        MR(i,t2)=1;
        LMR(t2)={'Sink'}; % uses SinkPattern for matrand2Pattern

    end
end
% eliminate selfloops in source and sink
MR(t1,t1)=0;
MR(t2,t2)=0;

% provide final out/in degrees for each node, the last two elements are source and sink (in this order)
sumColumn = sum(MR);
sumRow = sum(MR');

% alternative labellings. Here only illustrated for numbers as labels
rndnumber=ceil(Lsz.*rand(sz,1)); % Lsz --> numbers between 1 and Lsz can be labels
%rndlabel = cell(sz,1);
for j=1:sz
    LMR(j,1)={num2str(rndnumber(j))};
end
```

```

function [Match,uMatch]=rndMatchModel(szM,LszM,adjP,LP)
%szM : size of the graph model
%szP : size of the graph pattern

% Generates random graph model and uses given pattern (adjP and LP are the
% adjancency matrix and labels for pattern elements)
% The function does the pattern matching step
% and writes the matches on the graph model

%added LszM and LszP 01 Dic. (it allows labelling with numbers between 1 and LszX, X=P,M)

[MR,LMR]=matrand2(szM,LszM);
% % [PR,LPR]=matrand2(szP,LszP);

PR = adjP;
LPR = LP;

%directed
Match=MatchPattern(MR, LMR, PR, LPR);
%add label with 'label' and weight of the 'match'
S_Match=cell(length(Match),1);
Match_labels=cell(length(Match),1);
for i=1:length(Match)
    S_Match(i)={num2str(Match(i))};
    Match_labels(i)=strcat(LMR(i),'--',S_Match(i));
end

graph\_to\_dot5(MR,'filename','MR.dot','node_label',LMR);
graph\_to\_dot5(PR,'filename','PR.dot','node_label',LPR);
graph\_to\_dot5(MR,'filename','MatchResult.dot','node_label',Match_labels,'matches',Match);

%undirected
[uMR, uLMR]=GetUndirectedGraph(MR, LMR);
[uPR, uLPR]=GetUndirectedGraph(PR, LPR);
uMatch=MatchPattern(uMR, uLMR, uPR, uLPR);
%add label with 'label' and weight of the 'match'
uS_Match=cell(length(uMatch),1);
uMatch_labels=cell(length(uMatch),1);
for i=1:length(uMatch)
    uS_Match(i)={num2str(uMatch(i))};
    uMatch_labels(i)=strcat(uLMR(i),'--',uS_Match(i));
end
graph\_to\_dot5(uMR,'filename','uMR.dot','node_label',uLMR);
graph\_to\_dot5(uPR,'filename','uPR.dot','node_label',uLPR);
graph\_to\_dot5(uMR,'filename','uMatchResult.dot','node_label',uMatch_labels,'matches',uMatch);
```

```

```

```
function matches_to_dot(M,LM,P,LP)

Match=MatchPattern(M, LM, P, LP);
%add label with 'label' and weight of the 'match'
S_Match=cell(length(Match),1);
Match_labels=cell(length(Match),1);
for i=1:length(Match)
    S_Match(i)={num2str(Match(i))};
    Match_labels(i)=strcat(LM(i),'-',S_Match(i));
end

graph\_to\_dot5(M,'filename','M.dot','node_label',LM);
graph\_to\_dot5(P,'filename','P.dot','node_label',LP);
graph\_to\_dot5(M,'filename','Matches.dot','node_label',Match_labels,'matches',Match);

%undirected
[uM, uLM]=GetUndirectedGraph(M, LM);
[uP, uLP]=GetUndirectedGraph(P, LP);
uMatch=MatchPattern(uM, uLM, uP, uLP);
%add label with 'label' and weight of the 'match'
uS_Match=cell(length(uMatch),1);
uMatch_labels=cell(length(uMatch),1);
for i=1:length(uMatch)
    uS_Match(i)={num2str(uMatch(i))};
    uMatch_labels(i)=strcat(uLM(i),'-',uS_Match(i));
end
graph\_to\_dot5(uM,'filename','uM.dot','node_label',uLM);
graph\_to\_dot5(uP,'filename','uP.dot','node_label',uLP);
graph\_to\_dot5(uM,'filename','uMatches.dot','node_label',uMatch_labels,'matches',uMatch);
```

```

function graph_to_dot5(adj, varargin)

%show match

% graph_to_dot(adj, VARARGIN)  Creates a GraphViz (AT&T) format file representing
%                       a graph given by an adjacency matrix.
%  Optional arguments should be passed as name/value pairs [default]
%
%   'filename'  -  if omitted, writes to 'tmp.dot'
%  'arc_label'  -  arc_label{i,j} is a string attached to the i-j arc [""]
% 'node_label'  -  node_label{i} is a string attached to the node i ["i"]
%  'width'      -  width in inches [10]
%  'height'     -  height in inches [10]
%  'leftright'  -  1 means layout left-to-right, 0 means top-to-bottom [0]
%  'directed'   -  1 means use directed arcs, 0 means undirected [1]
%
% For details on dotty, See http://www.research.att.com/sw/tools/graphviz
%
% by Dr. Leon Peshkin, Jan 2004      inspired by Kevin Murphy's  BNT
%    pesha @ ai.mit.edu /~pesha

node_label = [];
arc_label = [];   % set default args
width = 10;
height = 10;
leftright = 0;
directed = 1;
filename = 'tmp.dot';
% added
matches = zeros(length(adj)); %till here
% added
edgeColor = zeros(length(adj)); %till here


for i = 1:2:nargin-1              % get optional args
    switch varargin{i}
        case 'filename', filename = varargin{i+1};
        case 'node_label', node_label = varargin{i+1};
        case 'arc_label', arc_label = varargin{i+1};
        case 'width', width = varargin{i+1};
        case 'height', height = varargin{i+1};
```

```
        case 'leftright', leftright = varargin{i+1};
        case 'directed', directed = varargin{i+1};
        case 'matches', matches = varargin{i+1}; % added
         %till here
        case 'edgeColor', edgeColor = varargin{i+1}; % added
         %till here

    end
end
fid = fopen(filename, 'w');
if directed
    fprintf(fid, 'digraph G {\n');
    arctxt = '->';
    if isempty(arc_label)
        labeltxt = '';
        color = '[color="green"]'; %added
    else
        labeltxt = '[label="%s"]';
        color = '[color="green"]'; %added
    end
else
    fprintf(fid, 'graph G {\n');
    arctxt = '--';
    if isempty(arc_label)
        labeltxt = '[dir=none]';
        color = '[color="green"]'; %added
    else
        %labeltext = '[label="%s",dir=none]';
        labeltxt = '[label="%s",dir=none]';
        color = '[color="green"]'; %added
    end
end
fprintf(fid, 'center = 1;\n');
fprintf(fid, 'size=\"%d,%d\";\n', width, height);
if leftright
    fprintf(fid, 'rankdir=LR;\n');
end

Nnds = length(adj);
for node = 1:Nnds               % process NODEs
    if isempty(node_label)
        fprintf(fid, '%d;\n', node);
    else
        %% added
        if matches(node)≠0
            fprintf(fid, '%d [ label = "%s", style ="%s", color ="%s" ];\n', node,
            node_label{node},'filled','green');
        end
        %% till here
        fprintf(fid, '%d [ label = "%s" ];\n', node, node_label{node});
    end
end



edgeformat = strcat(['%d ',arctxt,' %d ',labeltxt,';\n']);
edgeformat2 = strcat(['%d ',arctxt,' %d ',labeltxt, color,';\n']);

for node1 = 1:Nnds              % process ARCs
    if directed
        arcs = find(adj(node1,:));         % children(adj, node);
    else
        arcs = find(adj(node1,node1+1:Nnds)) + node1; % remove duplicate arcs
    end
    for node2 = arcs
        if isempty(arc_label)     % thanks to Nicholas Wayne Henderson nwh@owlnet.rice.edu
            if edgeColor(node1, node2)==1 % added
                fprintf(fid, edgeformat2, node1, node2); %% added
            else %added
                fprintf(fid, edgeformat, node1, node2);
            end
        else
            fprintf(fid, edgeformat, node1, node2, arc_label{node1,node2});
        end
    end
```

```
118 end
119 fprintf(fid, '}');
120 fclose(fid);
```

```
1
2  function GraMat\_to\_GraML(adj, attrName, attrType, attr, varargin)
3
4  % adj is the graph adjacency matrix
5  % attrName is a vector(n,1) with n entries representing the names of
6  % attributes for each graph vertex (e.g. "Label", "Cost",
7  % "Duration", etc. --> attrName={'nodeID';'id'; 'Label'; 'Cost'; 'Duration'})
8  % attrType is a vector(n,1) with n entries representing the types of
9  % attributes for each graph vertex (e.g. "int", "string", etc.
10 % attr is a cell matrix with the values of attributes for each vertex,
11 % attr's rows represent the vertices of the graph, and attr's columns
12 % represent the values of the attributes.
13 % e.g. attr={1 11 'A' 20 50; 2 22 'B' 30 60; 3 33 'C' 40 70}
14 %
15 % GraMat\_to\_GraML(adj, attrName, attrType, attr, 'filename','pattern.xml')
16 %
17 % GraphML is XML a graph format. See details here
18 % http://graphml.graphdrawing.org/
19 %
20
21 filename = 'GraphML.xml';
22
23 for i = 1:size(varargin,2)
24     switch varargin{i}
25         case 'filename', filename = varargin{i+1};
26    end
27 end
28
29 fid = fopen(filename, 'w');
30
31 fprintf(fid, '<?xml version="1.0" encoding="UTF-8" ?> \n');
32 fprintf(fid, '<!--   An excerpt of an egocentric social network  --> \n');
33 fprintf(fid, '<graphml xmlns="http://graphml.graphdrawing.org/xmlns"> \n');
34 fprintf(fid, '<graph edgedefault="undirected"> \n \n');
35
36 fprintf(fid, '<!--  data schema  --> \n');
37
38 m=size(attrName);
39 [attrf,attrc] = size(attr);
40 nodeID = zeros(attrf,1);
41 attrNameT=attrName';
42 attrTypeT=attrType';
43 for i=1:m
44     ii = cell2mat(attrName(i));
45     jj = cell2mat(attrType(i));
46     if (strcmp(ii,'nodeID')==1)
47         nodeID = attr(:,i);
48         nodeID = cell2mat(nodeID);
49         attrNoID = attr;
50         attrNoID(:,i) = []; %attr / attrName / attrType without NodeID
51         attrNameT(:,i) = [];
52         attrTypeT(:,i) = [];
53     else
54         fprintf(fid, '<key id="%s" for="node" attr.node="%s" attr.type="%s" /> \n',ii,ii,jj);
55     end
56 end
57
58 fprintf(fid, '\n <!--  nodes  --> \n');
59 [attrNoIDf,attrNoIDc] = size(attrNoID);
60
61 for k=1:attrf
62 fprintf(fid, '<node id="%d"> \n',nodeID(k,1));
63     for j=1:attrNoIDc %changed attrf+1
64         arg1 = cell2mat(attrNameT(j));
65         arg2 = cell2mat(attrNoID(k,j));
66         arg2 = mat2str(arg2);
67         fprintf(fid, ' <data key="%s">%s</data> \n',arg1,arg2);
68     end
```

```
69  fprintf(fid, '</node> \n');
70  end
71
72  [adjr,adjc]=size(adj);
73  fprintf(fid, '\n <!--  edges  --> \n');
74  for i=1:adjr
75      for j=1:adjc
76          if adj(i,j)==1
77              fprintf(fid, '<edge source="%d" target="%d"></edge> \n',i,j);
78          end
79      end
80  end
81
82  fprintf(fid, '\n </graph> \n');
83  fprintf(fid, '\n </graphml> \n');
```

D.3 Graph Models and Samples

This section illustrates some of the sample patterns and graph models used for the experimental evaluation of pattern matching and discovery techniques. Directed random graph models of 10, 50, 100 and 1000 vertices are shown in Figures D.2 to D.5. Details of vertices are not relevant in these figures, the intention is to illustrate the overall structure of the generated random graph process models.

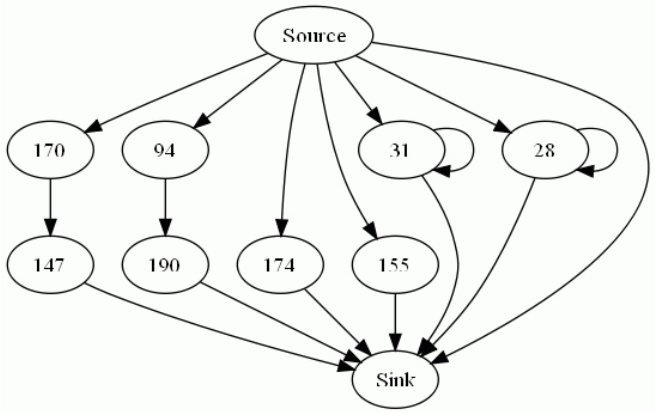


Figure D.2: Directed random graph model – ten vertices.

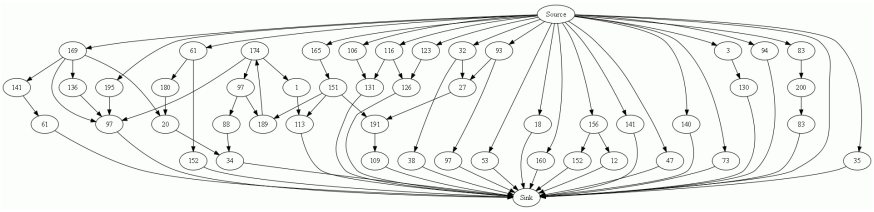


Figure D.3: Directed random graph model – fifty vertices.

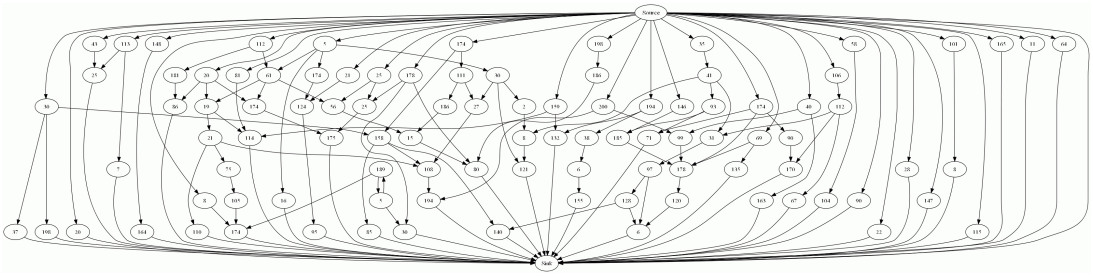


Figure D.4: Directed random graph model – hundred vertices.

Figure D.5: Directed random graph model – thousand vertices.

Appendix E

Interview Form

Questionnaire

Pattern Matching and Discovery for Process-centric Systems Design

Institution:

Software and System Engineering Group.

School of Computing, Dublin City University, Ireland.

General Instructions:

- Please complete with a cross in the field indicated as [ ] with the alternatives satisfy your opinion the most (i.e., you are allowed to mark more than one alternative).
- Answers to open questions can be written or recorded. If you prefer to be recorded please ask the interviewer before start the questionnaire.
- Please ask for clarifications anytime when answering the questionnaire, either if you don't understand something or you want to comment some issue related with the questionnaire.

Interviewee Profile (PART I)

Name (optional) _____

Company name (optional) _____

1. Role:

[ ] Business analyst
[ ] Business unit manager
[ ] Project manager
[ ] IT/enterprise/software architect
[ ] Software engineer / developer

Other:
_____
_____
_____

2. Organisation size:

[ ] Less than 50 employees
[ ] Between 51 and 250 employees
[ ] Between 251 and 500 employees
[ ] Between 501 and 5000 employees
[ ] More than 5000 employees

3. Organisation distribution:
[ ] Local to city
[ ] Local to country
[ ] Global

4. In a scale from 1 to 5, where 1 is "I am an expert" and 5 is "No knowledge/expertise", please refer to your knowledge about business process modelling

[ ]	[ ]	[ ]	[ ]	[ ]
1	2	3	4	5
(I am an expert)				(No knowledge/expertise)

5. In a scale from 1 to 5, where 1 is "I am an expert" and 5 is "No knowledge/expertise", please refer to your knowledge about any of the following concepts: process regulations / business process rules / process patterns / best practices / procedures.

[ ]	[ ]	[ ]	[ ]	[ ]
1	2	3	4	5
(I am an expert)				(No knowledge/expertise)

6. Consider the term *process constraints* as an umbrella term capturing the ideas behind process regulations (business process rules) / process patterns / best practices / procedures. Process constraints would restrict how processes are structured; their semantic and they identify constraints over data values associated to attributes of process elements.
Are you part of a team whose functions include analysing/designing organisation processes in regard to process constraints?

[ ] Yes
[ ] No
[ ] Occasionally

Other/Comments: _____

7. Please mark with a cross if you are familiar with any of the activities listed below. Note that listed activities are common activities performed during the analysis/design of processes regarding process constraints.

[ ] Process model verification
[ ] Process model transformation
[ ] Process model auditing
[ ] Process model comparison
[ ] Process pattern matching/identification
[ ] Process pattern discovery
[ ] Process model discovery

Please provide your comments regarding other activities performed during process analysis/design:

Business processes documentation (PART II)

8. In your organisation or the organisations you work(ed), what projects/activities generate graphical process documentation?

[ ] Process reengineering/improvement project
[ ] Process automation project
[ ] Analysis/design stages in IT project
[ ] Service-based architecture initiative
[ ] Process monitoring
[ ] Organisations merging
[ ] Organisations benchmark

Other:

9. In your organisation or the organisations you work(ed), in a scale from 1 to 5, where 1 is "only textual documentation" and 5 is "only graphical documentation", please refer to how processes are documented.

[ ]	[ ]	[ ]	[ ]	[ ]
1	2	3	4	5
(only textual)				(only graphical)
Go to (**13**)		Go to (**10**)		

Before Go to **13**/**10** → IF ANY, please score other organisation(s) you work(ed), and indicate their size and distribution. Size and distribution range is indicated in questions 2 and 3, respectively.
Organisation 2: score _____ size: _____ Distribution : _____
Organisation 3: score _____ size: _____ Distribution : _____

Comments:

10. What (graphical) process modelling notation are you used/using?

[ ] Business Process Modelling Notation (BPMN)
[ ] UML activity diagrams
[ ] Event-driven process chain (EPC)
[ ] IDEF 0 / IDEF 3
[ ] Some graphical notation for WS-BPEL

Other:
_____
_____
_____
_____
_____
_____

11. In your opinion, on average, how many tasks would you say a common (graphical) business process model contains?

[ ]	[ ]	[ ]	[ ]	[ ]
1	5	10	15	>15

Comments (if any):
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

12. In the context of your organisation, can you estimate how many graphically documented process models exist?

[ ]	[ ]	[ ]	[ ]	[ ]
1 to 4	5 to 20	21 to 50	51 to 100	> 100

IF ANY, please score other organisation(s) you work(ed), and indicate their size and distribution. Size and distribution range is indicated in questions 2 and 3, respectively.
Organization 2: score _____ size: _____ distribution: _____
Organization 3: score _____ size: _____ distribution: _____

Other organisations? (if any):
_____
_____
_____
_____
_____
_____
_____

Comments? (if any):
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

Business process constraints (PART III)

Remember that the term *process constraints* is considered as an umbrella term capturing the ideas behind process regulations / business process rules / process patterns / best practices / procedures. Process constraints would restrict how processes are structured; their semantic and they identify constraints over data values associated to attributes of process elements.

13. In your organisation, considering a scale where 1 is "daily basis" and 5 is "not at all", to what extent employees are required to use/satisfy *process constraints*?

[ ] 1 (daily basis) [ ] 2 [ ] 3 [ ] 4 [ ] 5 (not et al)

Go to (**15**) Go to (**14**)

Before Go to **14**/**15** → IF ANY, please score other organisation(s) you work(ed), and indicate their size and distribution. Size and distribution range is indicated in questions 2 and 3, respectively.
Organisation 2: score _____ size: _____ Distribution : _____
Organisation 3: score _____ size: _____ Distribution : _____

Other organisations? :
_____
_____
_____
_____

Comments (if any)? :
_____
_____
_____
_____
_____
_____

14. Please mention some factors you believe could influence the **non**-use/satisfaction of *process constraints*?
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

15. In your organisation or the organisations you work(ed), considering a scale where 1 is "strongly agree" and 5 is "strongly disagree",
Are there dedicated roles (and activities) for analysing/designing the organisation's processes in regard to *process constraints*?

[ ] 1 (strongly agree) [ ] 2 [ ] 3 [ ] 4 [ ] 5 (strongly disagree)

Go to (**16**) Go to (**18**)

Before Go to **16**/**18** → IF ANY, please score other organisation(s) you work(ed), and indicate their size and distribution. Size and distribution range is indicated in questions 2 and 3, respectively.
Organisation 2: score _____ size: _____ Distribution : _____
Organisation 3: score _____ size: _____ Distribution : _____

Comments? :
_____
_____
_____
_____
_____
_____

16. In your organisation or the organisations you work(ed), considering a scale from 1 to 5, where 1 is "only textual documentation" and 5 is "only graphical documentation", please refer to how *process constraints* are documented in relation to graphically documented processes? Remember that process constraints restrict aspects of processes.

[ ] 1 (only textual) [ ] 2 [ ] 3 [ ] 4 [ ] 5 (only graphical)

Go to (18) Go to (17)

17. What notation are you using (or used) to document *process constraints*?

[ ] (Enhanced) Business Process Modelling Notation (BPMN)
[ ] (Enhanced) UML activity diagrams
[ ] (Enhanced) Event-driven process chain (EPC)
[ ] (Enhanced) IDEF 0 / IDEF 3
[ ] (Enhanced) Graphical BPEL
[ ] Declarative/Rule/Constraint language (such as OCL, SWRL)

Other? :

_____
_____
_____
_____
_____
_____

18. What type of constraints *process constraints* impose over processes in organisations?

[ ] Structural constraints (ordering of process elements)
[ ] Constrains on data values associated to attributes in process elements
[ ] Time constraints
[ ] Constraints associated to the semantic of process elements

Others? / Comments? :

_____
_____
_____
_____
_____
_____
_____
_____

19. If you participate(d) in activities/projects involving the analysis/design of processes regarding *process constraints*, please mark with a cross if you performed any of the tasks listed below. Please also refer to other relevant activities in "other tasks?"

[ ] Process model verification
[ ] Process model transformation
[ ] Process model auditing
[ ] Process model comparison
[ ] Process pattern matching/identification
[ ] Process pattern discovery
[ ] Process model discovery

Other relevant tasks?

_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

20. If you participate(d) of activities/projects involving the analysis/design of processes regarding *process constraints*,
In a scale where 1 is "mostly automated" and 5 is "only manual", to what extent the tasks listed below were automated? Please mark [N/A] if you believe automation is not applicable to any of the tasks listed below.

[N/A] [1] [2] [3] [4] [5] Process model verification
[N/A] [1] [2] [3] [4] [5] Process model transformation
[N/A] [1] [2] [3] [4] [5] Process model auditing
[N/A] [1] [2] [3] [4] [5] Process model comparison
[N/A] [1] [2] [3] [4] [5] Process pattern matching/identification
[N/A] [1] [2] [3] [4] [5] Process pattern discovery
[N/A] [1] [2] [3] [4] [5] Process model discovery

Please also refer to other tasks – which are not been mentioned above – where you have been involved and if they have been automated to some degree:

Score: [1] [2] [3] [4] [5] Task: _____
Score: [1] [2] [3] [4] [5] Task: _____
Score: [1] [2] [3] [4] [5] Task: _____
Score: [1] [2] [3] [4] [5] Task: _____
Score: [1] [2] [3] [4] [5] Task: _____

21. For you organisation or other organisations you work(ed), in a scale where 1 is "strongly agree" and 5 is "disagree", Do you believe that you could benefit from *tools for automating* any of the tasks listed above or other tasks you frequently perform during the analysis/design of *process constraints*?, and Why?

[ ] 1 (strongly agree) [ ] 2 [ ] 3 [ ] 4 [ ] 5 (strongly disagree)

22. Assume a software tool would be able to *identify (match) and discover process patterns* in business process models, in general (not necessarily in your daily work), would you consider this a useful idea? Why?

Could this tool specifically benefit your work? How?

23. What **other** tasks during the analysis/design of processes regarding *process constraints* would you consider important to automate? Please provide comments for other tasks different to pattern matching and discovery.

24. Please feel free to openly comment about any aspects of analysis/design of processes in regard to *process constraints*.

WE THANK YOUR PARTICIPATION IN THIS INTERVIEW.

YOUR CONTRIBUTION WILL BE INCLUDED IN A STUDY OF TECHNIQUES AND TOOLS FOR AUTOMATED PROCESS PATTERN MATCHING AND DISCOVERY.

IF YOU ARE INTERESTED, YOU CAN RECEIVE OR ACCESS THE FINAL STUDY INDICATING YOUR INTEREST TO THE INTERVIEWER.