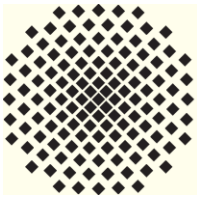


Universität Stuttgart

Process Fragments: Enhancing Reuse of Process Logic in BPEL Process Models

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Zhilei Ma

aus Qingdao, Shandong, Volksrepublik China

Hauptberichter:	Prof. Dr. Frank Leymann
Mitberichter:	Prof. Dr. Dimka Karastoyanova
Tag der mündlichen Prüfung:	23. Juli 2012

Institut für Architektur von Anwendungssystemen

der Universität Stuttgart

2012

Zusammenfassung

Die *Web Services Business Process Execution Language* (BPEL WSBPEL oder kurz) ist der Standard für die Erstellung von Geschäftsprozessen durch Orchestrierung von Web-Services. Allerdings ist die Modellierung von solchen Prozessen, besonders großen Prozessen, zeitaufwendig, fehleranfällig und daher teuer. Mit dem Wachstum der Anzahl und der Komplexität der Prozessmodelle in einer Organisation, wird Prozessmodellierung eine umfassendere Herausforderung.

Wiederverwendung (Reuse) wurde als ein wirksames Konzept etabliert, um Produktivität und Qualität in Software-Entwicklung zu steigern. Eine Teilprozesslogik wiederzuverwenden ist eine gewünschte Praxis in Modellierung von anderen Prozessen. Heute stellen Teilprozesse als das einzige granulare Instrument für die Wiederverwendung in Prozessmodellierung. Ein Teilprozess ist in der Regel wie ein in sich geschlossener Geschäftsprozess, der durch andere Geschäftsprozesse abgerufen wird und unterschiedliche Grade der Autonomie von der übergeordneten Prozesses besitzen kann. So ist die Wiederverwendung von beliebigen Teilen eines Geschäftsprozesses, insbesondere Teile, die nicht als eigenständige Prozesse gesehen werden können, zum Zeitpunkt des Schreibens dieser Arbeit noch nicht adressiert worden. Diese Art der Wiederverwendung ist besonders erwünscht bei der Erstellung von großen und komplexen Prozessen.

In dieser Arbeit konzentrieren wir uns auf die Wiederverwendung solcher Teile bei der Prozessmodellierung und präsentieren eine Reihe von Methoden für Modellieren, Extrahieren und Abfragen solcher wiederverwendbaren Teile. Wir nennen solche wiederverwendbaren Teile *Prozess-Fragmente*. Die Beiträge dieser Arbeit sind: (i) eine generische konzeptionelle Definition von Prozess-Fragmenten, einschließlich einer generischen mathematischen Definition von Prozess-Fragmenten basiert auf die Graphentheorie; (ii) eine formale Definition von BPEL-Fragmenten, die zeigt, wie das generische Konzept von Prozess-Fragmenten mit BPEL realisiert werden kann; (iii) einen Ansatz zum Extrahieren ausgewählten Aktivitäten als ein BPEL-Fragment aus einem BPEL-Prozess; (iv) einen generischen Graphen-basierten Algorithmus zum Abfragen von BEPL-Prozessmodellen und -Fragmenten.

Abstract

The Web Services Business Process Execution Language (WSBPEL or BPEL for short) is the standard for creating processes by orchestrating Web services. However, modeling processes, particularly large processes, is time-consuming, error-prone and therefore costly. With the growth of the number and complexity of process models in an organization, process modeling becomes a more comprehensive challenge, because it is cumbersome and not necessary for users to model every new process from scratch.

Reuse has been proven to be an effective concept to improve productivity and quality in software development. With the maturing of business process management technologies, reuse in business process modeling becomes one of the important research topics in the academia and industrial communities for business process management. Reusing a piece of process logic in other processes is a desired practice based on the case studies reported in the literature.

Today, subprocesses represent the only granule of reuse. However, subprocesses impose re-strictions on the syntactic and semantic completeness of the enclosed process logic. A sub-process is in general like a self-contained business process, but invoked by another business process and exhibits different degree of autonomy from the parent process. Thus, reuse of arbitrary parts of a business process, especially parts that cannot be seen as self-contained business processes, has not been addressed at the time of writing of this thesis. That kind of reuse is especially desired when creating large and complex processes.

In this thesis we focus on reuse of such parts during process modeling and present a set of methods for specifying, extracting, and querying such parts of a business process and enable their reuse. We call such arbitrary parts for reuse *process fragments*.

The contributions of this thesis are: (i) a generic conceptual definition of process fragments including a generic mathematical definition of process fragments based on a graph view; (ii) a formal definition of BPEL fragments, which shows how the generic process fragment concept can be realized within BPEL; (iii) an approach for extracting selected activities as a BPEL fragment out of a BPEL process; (iv) a generic graph-based algorithm for querying structural information of BEPL process models and fragments.

Acknowledgement

In the past six years I have received great support and encouragement, which made this work possible.

Prof. Dr. Frank Leymann has been a great doctor father, boss, and friend. His visionary insights in the research areas and professional guidance made this a thoughtful and rewarding journey. I have also learned a lot from his rich industry experience from which I still benefit in my professional life.

I would also to thank Prof. Dr. Dimka Karastoyanova as my second supervisor for her time and effort in supervising my thesis. Thank you also for the inspiring conversations that encouraged my research work and private life.

Furthermore I would like to thank my colleagues at the Institute of Architecture of Application Systems (IAAS) for the great discussions and working time in different projects and teaching activities.

My especial thank goes to my colleague and friend Branimir Wetzstein. Without the numerous constructive discussions and great time of co-writing many publications I could hardly imagine the progresses I made in the last years. Also his encouragement and sharing made this journey more joyful.

Last but not least I would like to thank my family for their support and understanding. They could always give me energy and confidence especially in the final stage of the doctorate. My sincere thanks also go to my friends, even though I could not mention each individual name here. Thank you all for the support in different kind of ways and the great time together.

Contents

List of Figures

Chapter 1
Introduction

This chapter provides a general overview of the research challenges addressed in this thesis. In Section 1.1 we motivate our research by identifying the inadequate capability in the current business process modeling approaches to reuse pieces of existing process models. In Section 1.2 we raise the research questions based on the motivation and outline our main contributions to solving them. Section 1.3 closes this chapter by giving a short description on the structure of this thesis.

1.1 Motivation

Business process management is one of the most crucial topics on the agenda of companies which want to improve their strategic competitiveness. These companies are challenged to optimize and redesign their business processes in a continuous and more agile manner [69]. A business process model is a formal description of a business process in the real world, which contains a set of interrelated activities and specifies the control and data dependencies between them. Such formalized process models[1] provide a means for a better communication between professionals in the business world, e.g. business analysts,

[1] In case of unambiguity we use the term *process* and *process model* interchangeably as the short form of *business process model*.

managers, and professionals in the IT world, e.g. software architects, developers[2]. IT-driven Business Process Management is a maturing discipline that includes concepts, methods, formalisms, and techniques to support the modeling, configuration, execution, and analysis of such processes [156]. In this way processes are no longer treated as static and isolated paper work, but rather as valuable information resources [93], which can be managed with help of Workflow Management Systems [95].

Process modeling has been established not only as the initial phase in the business process management lifecycle [5, 157], but also as an essential constituent in modeling modern enterprise architectures [121, 161]. In service oriented design paradigm [50] a process can be modeled as an orchestration [96] of the Web services [15, 68], which is, in combination with complementary techniques from Web service [155], the dominant implementation technology for Service-based Applications. The Web Services Business Process Execution Language (WSBPEL or BPEL for short) [16] is the standard for creating processes by orchestrating Web services. However, modeling processes, particularly large processes, is time-consuming, error-prone and therefore costly [95]. With the growth of the number and complexity of process models in an organization, process modeling becomes a more comprehensive challenge, because it is cumbersome and not necessary for users to model every new process from scratch.

Reuse has been proven to be an effective concept to improve productivity and quality in software development. With the maturing of business process management technologies, reuse in business process modeling becomes one of the important research topics in the academia and industrial communities for business process management. Reusing a piece of process logic in other processes is a desired practice based on the case studies reported in the literature [25, 123, 132].

Today, subprocesses represent the only granule of reuse. However, subprocesses impose restrictions on the syntactic and semantic completeness of the enclosed process logic. A subprocess is in general like a self-contained busi-

[2] For the purpose of simplicity and consistency, we use the term *users* as the umbrella term for these different stakeholders in this thesis.

ness process, but invoked by another business process and exhibits different degree of autonomy from the parent process. Thus, reuse of arbitrary parts of a business process, especially parts that cannot be seen as self-contained business processes, has not been addressed at the time of writing of this thesis. That kind of reuse is especially desired when creating large and complex processes. In this thesis we focus on reuse of such parts during process modeling and present a set of methods for specifying, extracting, and querying such parts of a business process and enable their reuse. We call such arbitrary parts for reuse *process fragments*.

1.2 Contributions

The term *process fragment* is used differently in different domains: for modeling collaborative processes, Lindert et al. [98] use process fragment as a collection of activities for which an organizational unit is responsible; for verifying process models, Vanhatalo et. al. use process fragment as a non-empty sub-graph of a workflow graph that is bordered by a single entry or single exit (SESE) edge or node [149]; for distributed execution, Khalaf [78] uses process fragment as a partition in the original process model augmented with additional interaction activities for retaining the original execution semantics. When analyzing existing research on process fragments, the reuse aspect of process fragments still needs to be addressed. Eberle et. al. [49, 48] introduce the concept of process fragment as incomplete process knowledge which can be dynamically stitched together at runtime. While their research focuses on the composition of process fragment, the research in this thesis underlines the concept, the application on BPEL, the extraction methods, and the query mechanism of process fragments.

To provide a common and consistent understanding of the reuse aspect of process fragments, we provide a conceptual framework for process fragments from a technology-independent view. We introduce a textual definition of process fragment from the process view. The definition from the process view emphasizes the intent and content of process fragments. It allows designing

process logic that does not represent self-contained process logic, but can be reused to model other processes. Furthermore, we provide a thorough analysis of the distinctions between the concepts of process fragments and subprocesses. The distinctions makes clear that our concept of process fragments enables encapsulating arbitrary process logic for reuse, while subprocess allows containing self-contained process logic.

In addition to the process view, we introduce a mathematical definition of process fragments based on a graph view, which is independent of BPEL. In the graph representation of a process fragment, the numbers of entry and exit nodes of a process fragment graph characterize the shape of the process fragment. Our definition on process fragment graph does not limit to Single-Entry-Single-Exit (SESE) process fragments, but enables designing reusable process fragments in arbitrary shapes.

Besides the definitions we have identified and classified different reuse styles. A reuse style specifies which parts of a process fragment can be manipulated when reusing it. In order to support applying the concept of process fragments to enhance reuse of arbitrary process logic in business process modeling, we introduce a lifecycle for process fragments [99]. The lifecycle guides users in applying process fragments in process modeling. On the other hand it also guides software vendors in providing a better tooling support.

After establishing the conceptual framework of process fragments, we need to show how the concept of process fragments can be applied to concrete process modeling languages. BPEL has been accepted as the standard for modeling processes by orchestrating Web services. It gains increasing popularity and applications both in academia and industries. For that reason we choose BPEL as the underlying process modeling language for the implementation of our reuse concept on process fragments. The BPEL fragment modeling language relaxes the strict syntax of standard BPEL and introduces new modeling constructs such as $< bag >$ activity to enable specifying arbitrary reusable process logic.

A BPEL fragment can either be modeled from scratch or extracted from existing BPEL process models. The BPEL fragment modeling language enables process modelers to model reusable BPEL fragments from scratch. To extract

a BPEL fragment from an existing BPEL process, we introduce in this thesis a mechanism which constructs the BPEL fragment based on the selected activities of the process modeler. Process modelers have options to let the extraction retain or completely eliminate the original control dependencies of the selected activities.

Efficient discovery of process fragments is of great importance: according to lessons learned from the practices in reuse history, users tend to build from scratch rather than to make the effort to find reusable artifacts [85]. We present a novel query mechanism which discovers BPEL fragments and process models that have similar process structures as the query request.

In summary, the novel contributions we provide within this thesis are:

- An analysis of general requirements of process fragments for reuse;
- A generic conceptual definition of process fragments;
- A generic mathematical definition of process fragments based on a graph view;
- A lifecycle for integrating process fragments into current process modeling approaches;
- A formal definition of BPEL fragments, which shows how the generic process fragment concept can be realized within BPEL;
- An approach for extracting selected activities as a BPEL fragment out of a BPEL process;
- A generic graph-based algorithm for querying structural information of BEPL process models and fragments.

1.3 The Structure of the Thesis

The content of this thesis is organized as follows:

Chapter 1, the current chapter, motivates the research topic on process fragments for reuse and outlines our contributions.

Chapter 2 gives an overview of the related work in reuse concepts in process modeling.

Chapter 3 introduces a generic concept of process fragments for reuse, pro-

vides a thorough analysis of the major characteristics of process fragments, a comparison of the concept of process fragments with the current established reuse approaches in process modeling, and a lifecycle for integrating process fragments into current process modeling approaches.

Chapter 4 presents the BPEL fragment modeling language. In this chapter we first analyze the requirements of BPEL fragment modeling language, especially which kinds of reusable BPEL process logic the language should support. Based on the analysis we introduced the syntax of BPEL fragment modeling language.

Chapter 5 provides the methods for extracting selected activities as a BPEL fragment. During the extraction we use opaque activities as temporary placeholders to retain the original control dependencies of the selected activities. Thus, we also present an approach of reducing such artificially generated opaque activities.

Chapter 6 describes a framework for mapping a BPEL process model to a directed acyclic graph, which defines a mathematical model used for the query algorithms.

Chapter 7 presents a generic graph-based algorithm for querying structural information of BEPL process models and fragments. This algorithm is generic enough so that it can be applied to query process models or fragments described in other process modeling languages and notations, as long as the process models and fragments can be transformed into a directed and acyclic graph.

Chapter 8 shows the architecture and prototypical implementation of a modeling tool with extended features for modeling and extracting BPEL fragments and a BPEL repository for storing and retrieving BPEL process models and fragments.

Chapter 9 concludes the thesis by giving a summary and identifying related future work.

Chapter 2
Related Work

In this chapter we study the existing work that is related to the research topics of this thesis.

2.1 Reuse in Process Modeling

Reuse has been established as a proven concept and technique to improve the productivity and quality in software development, e.g. in object-oriented programming [17, 61, 73], in component-based software development [7, 52, 70, 91, 107, 118]. Due to the poor outcome and benefit of ad hoc reuse, the reuse community came to the consensus that to realize the promise of reuse a systematic approach is needed.

Systematic reuse is the practice of reuse according to a well-defined repeatable process [105]. It is in general a technique to address the need for improvement of productivity, and quality and contributes to economic benefits on cost reduction [85, 97, 112, 113]. Productivity can be improved by allowing users to reuse existing knowledge and to develop assets like codes, models, documents and so on instead of recreating these artifacts [22]. Quality can be improved by well-designed process that guides the reuse of best-practice or proven software assets [59, 158].

Workflow [95] and service oriented process management [50, 156] intrinsically aim to boost reuse of software services in an interoperable and flexible manner. Software components or modules are exposed as services, which can be reused in different process models to collectively accomplish a business task or activity. Well prescribed process models are reused multiple times to instantiate the same process based on different context data. Especially, process models that are specified using established standards [94] can be reused across runtime platforms that are standard-compliant. We call this kind of reuse *re-apply*.

However, reuse in process management is not limited to re-applying the same process model multiple times, but also to enable process modelers to exploit existing process modeling artifacts in creating new business process models. In other words, reuse in process modeling is using a subset of existing process modeling artifacts to create new process models. In this thesis we focus on this kind of reuse.

2.1.1 Reuse by Customization

Process modelers may already practice reuse in their everyday work. When creating a new process model, a process modeler may take an existing process model and modify it until it meets the current requirements. This approach is a very primitive way of reuse and requires a lot of manual intervention. Thus, it is time-consuming, error-prone, and leads to a lot of repeated work.

As a way to overcome these shortcomings the concept of reference modeling has been introduced [89, 54]. A reference model is a process model that represents the general process knowledge shared within a certain domain [23, 130]. Reference process models aim to increase productivity by facilitating the reuse principle. Process models are created based on the best-practices or standards for a certain domain, which can then be customized to meet different application requirements. Examples of reference process models are SAP's reference models [76] and ARIS reference models for business process management [130].

Since reference process models normally contain information for multiple application scenarios, they should be projected to filter out the information that is not relevant in the current application context. Becker et al. [24] proposed a concept on how to configure multi-perspective process models. The different perspectives include application, document, data, and organizational unit. Process modelers can choose to hide or show a certain perspective of the process model for information filtering. A similar approach is presented in [88].

Gottschalk et al [65] developed a set of process configuration operators for *hiding* or *blocking* certain process activities. Hiding means that the execution of the affected activity is skipped, but the corresponding path is still taken. Blocking also disables the execution of the affected activity and the corresponding path cannot be taken anymore. Rossmann et al. extended the Event-driven Process Chains (EPCs) and presented the Configurable EPCs (C-EPCs) as an extended reference modeling language [124].

Karastoyanova proposed an approach on parameterization of workflow models to improve their reusability and flexibility [75]. The approach is based on the concept of workflow templates, which contain configurable parameters that can be customized either at design time or at runtime. A parameter could be an activity, an activity type, a transition condition on a control connector, a variable, or even a partner service. Semantic Web Service technologies can be utilized for (semi-)automatic substitution of parameter values. This approach improves the reusability of the process templates at design time and the flexibility of execution at runtime.

A lot of research has been conducted regarding reference process modeling [54, 55, 56, 139], process templates [89, 77], and configurable process modeling [109, 111]. Those approaches improve to certain degree the reusability of process modeling artifacts. But in general, process models must be reused as a whole. Enabling parts of business processes that can be reused in many different places or business contexts is a desired practice based on the authors' experience and case studies reported in the literature [25, 123, 132]. Modularized reuse is especially desired for complex or large business processes.

2.1.2 Reuse by Assembly

Patterns provide a means for abstracting and tackling recurring problems in a given domain [14, 37, 119, 122]. Patterns have been used in different areas of information technology, ranging from software design [61], to software architecture [38, 58], to enterprise application integration [71]. Research in process modeling has revealed several reuse approaches to improve the efficiency and quality of process modeling.

The workflow patterns: control flow patterns [128], workflow data patterns [127], workflow resource patterns [126], and exception handling patterns [125] aim to lay the groundwork and establish a common understanding on the requirements on process modeling languages and process-aware information systems. They provide a means for evaluating the expressiveness of various business process modeling languages and the capabilities of diverse workflow management systems. Workflow patterns can be considered as programming constructs in a conventional programming languages such as Java. The workflow patterns are fine-grained and there is no support for process modelers in how to apply these simple patterns in combination, which leads to many incorrectly modeled processes.

Gschwind et al. presented an approach of using compound control flow patterns to ensure the syntactic correctness of process modeling [67]. The authors developed a recommendation mechanism. The mechanism considers the modeled static process knowledge and known control flow patterns, based on which it makes suggestions on how to complete the partial workflow graph to make it a well-structured workflow graph.

Vanhatalo et al. [151] proposed another approach for computing a completion of a workflow graph. This approach can also be used to refactor existing workflows to transform them into well-structured workflows if necessary. A workflow graph is decomposed with the help of normal [150] and the refined process structure tree [149] into logically atomic parts. If a subtree does not consist of matching pairs of start and end node as specified in the control workflow patterns, then the modeling tool makes a completion suggestion or undertakes a refactoring process.

However, both of the approaches focus on the correctness of a workflow graph. The compound patterns do not contain process knowledge and are not sufficiently enriched with context information to represent a reusable building block for process modeling.

Lindert et al. proposed an approach of modeling of inter-organizational processes using the concept of *process model fragment* [98]. This approach aims to enable autonomy of the participating organizational units in process modeling. A process model fragment describes all the activities that an organizational unit has to perform during a given process. The respective organizational unit can autonomously describe and execute the process fragment, e.g. using its own workflow management approaches and systems. The notion of autonomy was derived from the work of Warnecke [154], which means that the organization unit must be a self-containing and self-organizing unit. Process model fragments can be connected through their interfaces to build up the whole process. There are two connection approaches: vertical and horizontal [98, 81]. Vertical connection approach allows refining a parent process fragment by a child process fragment. A child process fragment is a more detailed description of the parent process fragment. Horizontal connection approach can be considered as a composition of the interfaces of process fragments. To connect two interfaces of process fragments, the documents and events that the interfaces provide and consume have to match with each other. Organizing process activities based on their organizational affiliation is only one possible view to project on a process model. Beside this, a process model fragment must contain *all* of the activities that an organizational unit has to perform during a process. This constraint extremely restricts the flexibility to reuse arbitrary process logic. This paper primarily focuses on describing the methodology, but not on extracting and querying process fragment models.

Adams et al. presented an approach using *worklets* to enable dynamic flexibility and evolution of workflows during runtime [8, 9]. A worklet is defined as a smaller, self-contained and complete workflow process which handles one specific task in a larger and composite top-level process. The top-level process model captures the entire workflow at the macro level. Worklets are dynamically selected based on the context of each task in the macro process. Each

task in the macro process model can be linked to a catalog of worklets, each of which represents a possible implementation of the task. During process execution, the appropriate worklet is selected based on the context data of a particular process instance and the rules associated with the activity. The worklet is executed as a nested subprocess to the calling process.

Trickovic [146] analyzes three possible approaches for reuse of BPEL processes and came to the conclusion that subprocess [83, 95] seems to be the more general approach to solve this issue. BPEL-SPE [82] is a BPEL extension that allows users to define certain kinds of BPEL processes as subprocesses, which can be invoked by the same or another BPEL process.

Thom et al. introduced the concept of *activity pattern* [143, 144]. An activity pattern describes a business function that occurs frequently in different process models. The authors identified seven pattern variants based on their empirical study, e.g. notification pattern, decision pattern, and approval pattern. Each activity pattern is described using a pattern language, including description, example, problem, issues, and solution, to guide a process modeler in reusing the activity patterns.

Worklets, subprocess, and activity pattern all require that the encapsulated process logic must be a self-contained and complete workflow process. Worklets improve the flexibility of changing process at runtime, but do not address the flexibility of reuse existing modeling artifacts as building blocks at design time. Subprocesses and activity patterns are in general executable and may carry different grade of autonomy against their caller processes. To enable design for reuse and design by reuse to a more extended degree, process modelers need a more flexible approach, which should allow them to define arbitrary process logic as reusable building blocks, which could be syntactically and semantically incomplete for execution.

Görlach et al. introduces an extension of BPEL for modeling reusable compliance fragments [64]. A compliance fragment encompasses control and data flow that describe compliance constraints [134].

Researchers have also identified different scenarios in the domain of business process management, such as for business process compliance management [134, 136, 64], and for dynamic runtime flexibility [49].

2.2 Decomposition in Process Modeling

Process models can be decomposed into parts for different purpose. In this section we will study the existing work on decomposing process models.

Khalaf introduced a mechanism for splitting a BPEL process into several partitions [78]. Each partition can be outsourced to different participants and can be executed on process engines in a distributed manner. To outsource a part of a BPEL process a process modeler has to assign the activities to their respective partitions. The BPEL process is then spitted into disjoint partitions, each of which contains the activities that a certain participant (role) is responsible for [79]. Additional activities need to be added to each partition for passing instance data and control flow. These partitions are wired together through explicit data links defined in BPEL-D, which is a BPEL extension that transforms implicit data flow in BPEL into explicit ones by using data links. The wiring through data links ensures that each partition can get the data needed for distributed execution. A new coordination protocol type that is plugged into the WS-Coordination framework enables the coordinated and distributed execution of the partition including decomposed loops and scopes [80]. This guarantees that the distributed execution as a whole yields the same operational semantic as the original process. However, this approach still requires a central coordinator to be able to handle the execution of spitted scopes and loop constructs [104].

Wutke developed an approach for automatically splitting BPEL processes into logical segments in order to enable the execution of a BPEL process in a distributed and decentralized manner [160]. The partitioning mechanism consists of three phases. In the first phase, users can define to which partitions the respective message receiving activities (`<receive>` and `<pick>`) should be allocated. In addition, users can also specify static parameters, e.g. the variable *request* should always be assigned to the partition *receive request*. In the second phase, interaction activities (`<invoke>` and `<reply>`) are assigned. For each `<invoke>` activity the mechanism discovers first the registered services that provide equivalent functionalities and compatible interfaces as the `<invoke>` activity itself. The discovery takes also non-functional

properties specified by users with WS-Policy [21] into consideration. The service that results in the minimal number of partitions will be selected. Each <reply> activity will always be assigned to the partition to which the respective <receive> activity belongs. In the third phase, all the other activities in a BPEL process will be partitioned. The partitioning is conducted with the goal to achieve the minimal interaction between the partitions of the BPEL process. The partitioning mechanism does not physically split the BPEL process, but describes which activity belongs to which logical partition in the distributed deployment descriptor.

Automatic process partitioning can be supported by a cost function that calculates the optimum under consideration of user defined criteria [104]. Danylevych et al. proposed an approach for producing recommendations on how to split a process model into disjoint partitions so that the performance of the distributed execution of the partitions is improved [45]. A process model is considered as a global transaction, which can be partitioned into parts called stratified transactions. The authors developed a hybrid approach utilizing algorithms like hillclimbing and simulated annealing for transaction stratification to produce recommendations depending on the optimization criteria such as time, cost, etc.

While the previous approaches focus on decomposition of process models for distributed process execution, the following work concentrate on process modeling.

Vanhatalo presented an approach to decompose a workflow graph into single-entry-single-exit subgraphs [147]. An entry node is a node in the graph that has no incoming edge, while an exit node is a node in the graph that has no outgoing edges. At the heart of the approach are the normal [150] and refined process structure tree [149]. The normal process structure tree decomposes a workflow graph at the nodes, i.e. each node belongs to only a single subgraph. In contrary, the refined process structure tree decomposes a workflow graph at the edges, i.e. each edge belongs to exactly one subgraph while nodes may be shared between multiple subgraphs. Besides decomposition of workflow graphs the normal and refined process structure trees can also be applied to conduct control flow analysis [150], to refactor process models to ensure the

syntactic correctness [151], and to auto-complete a workflow graph with missing end nodes [151], e.g. gateways of a process model described in BPMN.

Decomposing each process model into single-entry-single-exit subgraphs and analyze whether they can be reused to create new process models could be time-consuming and expensive. Gerlach developed an approach to help process modelers discover candidates of reusable parts of existing process models [62]. The approach makes the assumption that recurring process structures could present reusable process knowledge in creating new process models. The mining of such recurring process structures has been reduced to the *frequent subgraph mining problem* [86]. To search for frequent subgraphs users have to define a minimal support: the minimal support specifies how many different process models in the repository have to contain the subgraph to make it a frequent subgraph. The implementation of the approach is based on the MARGIN algorithm [145] because of the better performance in comparison to other existing algorithms. The implementation works on a set of process models and returns a set of frequent subgraphs of the process models.

Decomposition makes complex process models better manageable, which is also one of the major motivations to generate process views. A process view can be considered as a projection on a process model which allows abstraction or hiding of undesired information [117].

Eshuis et al. introduced a two-step approach for constructing customized process views on process models that are specified using block-structured process languages [51]. In the first step selected activities are aggregated into a single activity. The defined rules for aggregation ensure the aggregated process view is acyclic. In the second step, process modelers can choose the activities in the aggregated process view that should be hidden or omitted for certain target groups, such as customer, partners, suppliers, etc.

Schumm et al. analyzed related works of constructing process views and categorized them into patterns of process views [135]. Their patterns are categorized into structure patterns, presentation patterns, inter-view patterns, and augmentation patterns. Structure patterns encompass the common operations for process model transformation, such as abstraction, aggregation, insertion, omission, and preservation. Presentation patterns include aspects such as ap-

pearance (color, size, etc.), layout, scheme, and theme (which information of the process should be shown). Inter-view patterns define how to combine views from the same process model (orchestration inter-view) and how to combine views from different process models (choreography inter-view). Augmentation patterns describe how a process view can be described using additional information, such as runtime information, calculated information, and other human-added annotations.

Similar concepts have been applied to different process modeling languages. Process views for BPEL [16] can be found in [163], for Event-driven Process Chains (EPCs) [60] can be found in [66], and for BPMN [4] can be found in [42]. Further research work can be found in [90, 142, 138].

2.3 Query in Process Modeling

Corrales et al. proposes a graph-based approach for comparing the operational semantics of two BPEL processes [43]. Each BPEL process model being compared is transformed into a graph representation, which a node represents an activity and the edge represents the control flow between the activities. In order to simplify the comparison of the interaction activities the graph transformation splits an `<invoke>` activity that models a synchronous interaction into two activities, i.e. an `<invoke>` activity (one way) and a `<receive>` activity. The matchmaking mechanism first examines whether the two graphs being compared are isomorphic. As subgraph isomorphism cannot always be found, process models that have approximate structure (see Section 7.2.2 Approximate Match) as the query graph should be found: the approximate matchmaking is based on the concept of error-correction (sub-)graph isomorphism, which computes the edit distance between the two process graphs being compared. An implementation can be found in [44]. Other approaches of measuring behavioral similarity between process models can be also found in [46, 47].

Beeri et al. developed a query language BP-QL for querying BPEL process models [26, 27, 28]. The language provides a graphical notation, which is similar to the popular graphical notations in commercial BPMN and BPEL

modeling tools. To query BPEL processes a user just models the query result using the graphical notations. With the graphical notations activities, data and their properties are modeled as nodes. Control flows are denoted by activity flow edges, while data flow edges specify the data flow between the respective activities. Property nodes are used to define global properties and are directly linked to the query process. The query can be formulated in two dimensions: a path-based dimension allows to query possible execution paths in BPEL process models; a zoom-in dimension allows to transitively navigate into the orchestrated Web services and include them at any depth of nesting for the query processing.

The formulated query is translated into a specification using Active XML [6] for evaluation, which is a XML-based non-standard language for data integration. Neither with the proprietary graphical notation nor with Active XML process modelers can use BPEL process models or BPEL fragments as query request. The query evaluation of BP-QL can only return a complete BPEL process model or a subgraph of it that has a single entry node and a single exit node. This is also a limitation as the BPEL fragments defined in this thesis may have multiple entries and exits. Last but not least the time complexity of evaluating a BP-QL request is exponential in the order of the query graph.

Awad introduced another graph-based query language BPMN-Q for structural querying BPMN process models [18]. In BPMN-Q the author allows process modelers to use some standard BPMN modeling elements, such as tasks, events, gateways, and sequence flow. In order to improve the expressiveness of the query language, the author has also introduced some new modeling elements, including a variable activity, generic split and join gateways, and a generic node construct. The query request will be evaluated against a process model repository. To reduce the search space the query processor starts with identifying the relevant process models to be compared. A relevant process model must contain all the query nodes in its process graph [20]. The query processing is based on path extraction, which computes paths with all lengths between any pair of adjacent query nodes.

As the node set of the process graph must be a superset of the activity nodes in the query graph, the query processor is not able to return process models that

contain only a subset of the activity nodes in the query graph. In addition, computing similarity based on path extraction cannot precisely reflect the similarity between the complete process graph and the query graph (cf. Section 2.4).

Awad et al. extends the query language BPMN-Q [18] by employing an ontological dimension in the query processing [20]. To avoid manual annotations of the process models the approach exploits the *enhanced Topic-based Vector Space Model* (eTVSM) ([87, 116]. eTVMS uses its pre-defined ontology to compute semantic similarities of the task labeling and transforms them into a numeric similarity value. Thus, this approach uses only information that is already available in the process model being compared and does not need additional annotations. The implementation of BPMN-Q can be found in [129]. In [19] the authors provide another application of BPMN-Q for compliance checking of process models.

Markovic et al. proposed a method for querying ontologically annotated process models [103, 101]. To enable an expressive query of process models each process model should be annotated with ontological concepts [102], including functional, behavioral, organizational, and informational perspectives. The ontology framework is specified using Web Service Modeling Language (WSML) [140]. A query request may comprise two parts: query request on static properties is specified using WSML logical expressions, while query request on process behavior is specified as process model definition. Querying on static properties is performed first to narrow down the search space of querying on dynamic behavior, as the latter is more expensive than querying static properties. The authors use ontologized π-calculus to describe the query request, which is then compared using congruence and bisimulation properties with process models stored in a process repository [101].

Vanhatalo developed a repository for BPEL process models [148]. Each BPEL file is annotated with organizational specific metadata. Through the repository API users can manipulate BPEL files as Java objects and query BPEL processes using the Object Constraint Language (OCL). Another repository for BPEL processes and fragments can be found in [133]. The query mechanism operates on the metadata of process fragments, such as fragment names, keywords, the number of the entries and exits of a process fragment, etc. In

addition, users can integrate specific query processors through an extension interface of the query engine, e.g. query processor of WS-Policies that are associated with the process fragments. The structural aspect has not been addressed in both of the repositories.

2.4 Similarity Measurement

An essential issue in discovering approximate matches based on a graph-based matchmaking mechanism is how to measure the similarity of two graphs that are being compared. Different approaches have been developed for that purpose, such as edit-distance-based, path-based, index-based, and common-subgraph-based similarity measurement approaches. In following we discuss each of these approaches in detail.

Similarity Measurement based on Graph Edit Distance

Graph edit distance is a common way to compute the similarity between two graphs in approximate graph matchmaking approaches [31, 34, 53]. A Graph G_1 can be transformed into another graph G_2 by applying edit operations such as insertion, deletion, and substitution of nodes and edges. Thus, the edit distance of the two graphs G_1 and G_2 is indicated by the shortest sequence of edit operations that transforms G_1 into G_2 [34]. In addition, the author defines a special cost function f on the edit operations. Under this cost function f, any deletion and insertion operations on nodes has a cost equal to one. The deletion and insertion of an edge that are connected with a node that is also deleted or inserted respectively, has no cost. Substitutions of identical nodes and edges have zero costs, while substitutions of different labeled nodes and edges have infinite cost. Under this cost function, the author has proven that any function f with the minimum cost of edit operations for transforming a graph G_1 to G_2 is a graph isomorphism between a subgraph $\hat{G}_1$ of G_1 and a subgraph $\hat{G}_2$ of G_2, where both $\hat{G}_1$ and $\hat{G}_2$ are maximum common subgraphs of G_1 and G_2. Therefore, an algorithm that computes the graph edit distance can be used to

compute the maximum common subgraph if it runs under the cost function f introduced in [34].

It is well known that subgraph isomorphism computation is an NP-complete problem. Consequently, approximate subgraph isomorphism (aka error-tolerant subgraph isomorphism) is also in NP and even harder than subgraph isomorphism detection [108]. Based on this, we discuss whether there exist efficient algorithms for approximate matchmaking of special graphs, e.g. tree and DAG.

As tree is a special case of DAG, if there are no efficient algorithms for trees, then the approximate matchmaking problem of DAGs cannot be solved in an efficient manner. The embedding problem of the *exact ordered* tree has been studied in [120] and is solvable in polynomial time. However, we are interested in *unordered* embedding in trees, because the elements of a BPEL process in its corresponding XML tree do not have necessarily a fixed order [110]. For example, order of the parallel running activities of a `<flow>` activity is not defined. Furthermore, we are not interested in exact embedding that preserves the parent-child relationship, but the approximate matches that have similar structures as the query request. The approximate matchmaking for unordered trees is denoted as the *approximate nearest neighbor (ANN)* problem for unordered labeled trees [162]. The authors have proven that solving the ANN problem with edit distance is NP-complete. Therefore, we do not consider using edit distance for the similarity measurement for the approximate query processing. Other approaches for similarity measurement based on edit distance of two graphs can be found in [12, 31, 131].

Similarity Measurement based on Path Extraction

Due to the high complexity of solving the ANN problem using edit distance Shasha et al. proposed a new approach for approximate search of unordered labeled trees [137]. The authors measure the distance between the query graph Q and a process graph P by the total number of the root-to-leaf paths in Q that do not appear in P. The reasons to use path extraction for similarity measurement of two graphs are twofold: (i) the paths reflect the parent-child and ancestor-descendant relationships between the nodes, which is important for di-

rected graphs; (ii) the comparison of paths can be easily transformed to string searching problem so that one can exploit existing efficient string searching algorithms for both exact and inexact matchmaking. Query requests for inexact matchmaking can be formulated by using wildcard characters as nodes in the query graph. All matches of each query path are then merged together using structural join algorithms [13, 33, 40, 41] to produce the result.

This approach requires that each root-to-leaf path in the query graph must completely appear in the process graph being compared. That means a path in the process graph that contains only a proper subset of a path in the query graph will not be considered in the similarity measurement. Our algorithm also returns process models or fragments that partially contain nodes of a path in the query graph.

Similarity Measurement based on Maximal Common Subgraphs

Some approaches measure similarity of two graphs by computing the maximal common subgraph (mcs) of the two graphs. The similarity of graphs G_1 and G_2 can be computed as follows [36]:

$$s(G_1, G_2) = \frac{|mcs(G_1, G_2)|}{max(|G_1|, |G_2|)} \tag{2.4.1}$$

As the subgraph isomorphism problem, the maximal common subgraph problem is also a famous NP-complete problem [74]. Different algorithms for computing the maximal common subgraph have been presented in [92, 106, 35, 153]. Maximum common subgraphs can be used to compute the graph edit distance for error-tolerant graph matchmaking approaches [34, 36]. Like graph edit distance computing the maximal common subgraph requires exponential time and space due to the NP-completeness of the problem. Besides the high complexity, using maximal common subgraphs to compute the similarity of two graphs may not reflect their actual similarity. On the one hand, there may exist more than one maximal common subgraphs; on the other hand, the non-

maximal common subgraphs should also be taken into account to compute the similarity more precisely.

Chapter 3
Process Fragments for Reuse

This chapter presents the concept of process fragments for reuse. In Section 3.2 we introduce a textual definition of process fragments from the process view. Section 3.3 presents a mathematical definition of process fragments based on graph theory. Based on the definitions we discuss the shapes (Section 3.4), the granularities (Section 3.5), and the reuse styles (Section 3.6) of process fragments. In Section 3.7 we introduce the process fragments modeling lifecycle.

3.1 Introduction

In this chapter we introduce process fragments for reuse in a language- and notation-independent way. Later in this thesis, we will present how this concept can be applied to Business Process Execution Language (BPEL), which is an established standard for modeling business processes. As different process modeling languages and notations use inconsistent terminologies, we align our vocabulary with the Workflow Management Coalition Terminology & Glossary [3]. Despite our endeavor to keep a technology-independent view, using a particular notation helps to clearly illustrate the concept and to have a better comprehension of the concept.

We consider process fragment from two different perspectives: the process view and the graph view. The process view provides the concept that we will

apply to BPEL for defining BPEL fragments (see Cahpter 4). The BPEL definition of process fragments will be used for design and extraction of BPEL fragments (see Chapter 5). The graph view will be used to define the mapping of BPEL fragments to graphs (see Chapter 6) and for queying BPEL fragments (see Chapter 7) in order to use graph algorithms.

3.2 The Process View

Before starting with the definition of process fragments we examine the linguistic meanings of the term *fragment*. Merriam-Webster's Online Dictionary defines the term *fragment* as "*a part broken off, detached, or incomplete*" [2]. Collins Dictionary explains that a *fragment* is "*a piece broken off or an incomplete piece*" [1]. From the dictionary definitions we can infer two fundamental characteristics: **partial** and **incomplete**. As a specialization of *fragments*, process fragments inherit these two characteristics.

Definition 1. Process Fragment
A process fragment is a part of a process that must contain at least one activity and it may be syntactically and semantically incomplete. □

A very frequent use case of process fragments is that of a reusable granule, which is the focus of this thesis. Reuse in process modeling generally refers to using a part of existing process logic to create new process models. Process logic is encoded in process activities, the control and data flow between the activities rather than in other process modeling artifacts, such as definition of variables. For that reason, we require that a process fragment must contain at least one activity. An activity describes a piece of work that forms a logical step within a process [3].

A process fragment may be incomplete. In order to enable process modelers to design process fragments with high reusability we need to relax the rigid specification of the underlying process modeling languages or notations, which otherwise enforce completeness and correctness. Loosen these restrictions leads to syntactic and semantic incompleteness of process fragments.

Syntactic Incompleteness

A process fragment is syntactically incomplete if one enclosed process modeling artifact does not contain all the **mandatory** constituents specified in the original syntax of the underlying process modeling language or notation. In the following we show an example that uses BPEL as the underlying process modeling language.

With BPEL, conditional behavior can be modeled with an `<if>` activity. The syntax of BPEL requires that an `<if>` activity must contain at least one conditional branch, which in turn contains exactly one mandatory activity. Further conditional branches can be defined by using `<elseif>` constructs within the `<if>` activity. The activity container `<else>` within an `<if>` activity defines the default behavior if no conditional branch is taken.

A process modeler may want to define an `<if>` activity as a reusable process fragment, which contains only the `<else>` branch. The process fragment serves as a template for modeling `<if>` activities that share the same default behavior. Conditional branches can be added on demand at the time of reuse. Figure 3.1 illustrates the process fragment. As mandatory transition condition and primary activity of the `<if>` branch are missing, the `<if>` activity is not compliant with the BPEL syntax. We call such incompliance with the BPEL syntax caused by missing mandatory constituents the *syntactic incompleteness*. The `<if>` activity shown in Figure 3.1 is syntactically incomplete.

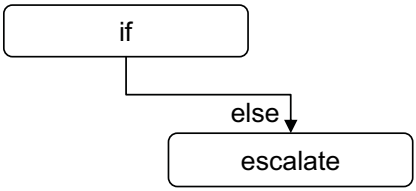


Fig. 3.1 An *if* activity that has only the *else* branch is syntactically incomplete according to the BPEL syntax.

Note that syntactic incompleteness is not equivalent to syntactic incorrectness. Syntactic incorrectness refers to an illegal usage of process constructs.

For example, according to the BPEL syntax it is illegal and incorrect to use a fault handler <catch> immediately in a <flow> activity. Syntactic incompleteness can be considered as a violation of the cardinality constraints defined in the underlying syntax. We can transform a syntactically incomplete process fragment into a syntactically complete one, if we augment the process fragment with the mandatory process elements. But a syntactically incorrect process fragment cannot necessarily be corrected in this way.

Semantic Incompleteness

A process fragment could also be semantically incomplete. Semantic incompleteness means that the essential modeling artifacts that are needed to capture the intended process logic are underspecified or missing. As shown next, two major indicators of semantic incompleteness are: (i) the presence of variability points and (ii) lack of adequate information on process context.

A variability point represents a placeholder of a delayed design decision [32]. In order to increase the reusability of a process fragment, process modelers may specify variability points to allow others to make the specific design decisions at the time of reuse. For example, a variability point can either be a placeholder for an activity or be a placeholder for the value of an attribute. Exemplary constructs that can be used as variability points in process models can be found in existing process modeling languages and notations. For example, an <opaqueActivity> in BPEL represents an explicit placeholder for exactly one executable BPEL activity; while the ##opaque token represents a placeholder for the value of an executable BPEL attribute.

Another major indicator of semantic incompleteness of a process fragment is the missing definitions of the elements that are used in a process activity. Let's consider the process fragment in the following listing as an example. The fragment contains only one <invoke> activity. Although the <invoke> contains no placeholder, the semantics of the activity is unclear. How is the partner link defined? Is the "*Provider*" for booking a car, a flight, or a hotel room? What kind of data does it need for the input variable and what kind of data do

we get as response? Without the definitions of these elements the semantics of the process fragment is ambiguous.

```
<invoke name="booking"
  suppressJoinFailure="yes"
  partnerLink="Provider"
  portType="PRO:Booking"
  operation="Book"
  inputVariable="request"
  outputVariable="getResponse">
```

Listing 3.1 The semantics of the invoke activity without definitions of partner link, port type, variables, etc. in BPEL is ambiguous.

Note that not all process modeling languages require process context as the mandatory constituent of a process model. For instance, correlation sets, partner links, message exchanges, and variables are not mandatory in a BPEL process. Thus, the missing definitions of these elements do not necessary lead to syntactic incompleteness. In contrast to that, omitting the definitions of these elements leads to semantic incompleteness, because the undefined elements make it difficult for both humans and machines to understand and execute process models correctly.

In the following chapters we will use the process view for defining BPEL fragments (Chapter 4) and for extracting BPEL fragments (Chapter 5). However, our query algorithm works on a generic graph model. To use the graph-based query algorithm (Chapter 7), we have to map a BPEL process model or fragment to a graph model (Chapter 6). For that reason we introduce the definition of process fragments from the graph view.

3.3 The Graph View

In a graph-based approach a process model is represented as a Directed Acyclic Graph (DAG) [94]. The graphical representation of a process model is also called a Process Model Graph (process graph for short) [78, 95]. In a process graph each node represents an activity in the given process model. The directed

edges represent the control connectors, which define the possible flow of control between the activities.

The ancestor-descendant relationship of two activities in a process graph indicates the causal or the temporal control dependency between them. If an activity a_1 is neither an ancestor of another activity a_2 nor a descendant of the activity a_2, then we say the activity a_1 is parallel to the activity a_2. In this case there exist no directed path between the activities a_1 and a_2.

Definition 2. Process Fragment Graph
A process fragment graph consists of one or more fragmental components of a process graph. Let $G = (V,E)$ be a directed and acyclic process graph. V is a set of nodes, where each node represents an activity. E is a set of edge, where each edge represents the control flow connectors between the activities. A fragmental component of G is a directed, acyclic, and weakly connected graph $G' = (V',E')$, where $V' \subseteq V$ and $E' \subseteq E$. For all $e' \in E'$ it satisfies one of the following conditions:

$$\begin{cases} \pi_1(e') = v'_1,\ \pi_2(e') = v'_2 & v'_1 \text{ and } v'_2 \in V' \\ \pi_1(e') = v'_1,\ \pi_2(e') = \perp & v'_1 \in V',\ \perp \text{ is undefined} \\ \pi_1(e') = \perp,\ \pi_2(e') = v'_2 & \perp \text{ is undefined},\ v'_2 \in V' \end{cases}$$

□

A fragmental component is weakly connected. For a directed graph (digraph) there are basically two views on connectedness: strict view and relaxed view.

The strict view defines connectedness based on the explicit definition through directed edges. A directed graph $G = (V,E)$ is *connected* if for each pair of distinct vertexes $u,v \in V$, there exists a directed path from u to v in the graph G [10]. Otherwise, we say the graph G is *disconnected*.

The relaxed view treats the connectedness of a digraph by examining its underlying non-directed graph. A directed graph $G = (V,E)$ is *weakly connected* [10] if its underlying non-directed graph is connected, i.e . for each pair of distinct vertexes $u,v \in V$, there exists a path (u,v) in the graph G. As Figure 3.2 shows, the directed graph P on the left side is not connected, because there does

not exist a directed B,C-path for the distinct vertexes pair (B,C) in P. However, P is weakly connected, as its underlying non-directed graph P' is connected.

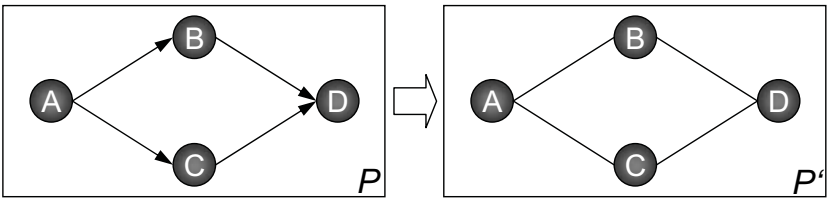


Fig. 3.2 The underlying non-directed graph P' of P is a weakly connected graph.

Consider the example illustrated in Figure 3.2. The subgraph P could be considered as a reusable process fragment even though it is not connected. Thus, we use the term *weakly connected* in the definition of process fragment graph. We say a directed graph is weakly connected, if its underlying non-directed graph is connected [10].

A fragmental component may contain edges that have either no start node or no end node. Incomplete incoming edges (with no start nodes) or incomplete outgoing edges (with no end nodes) can be used to model unknown process logic or retain the original structure when extracting a process fragment. An edge with neither start node nor end node is not allowed in our concept.

A fragment graph may contain one or more fragmental components. Consider the following example shown in Figure 3.3. The process model at the top of the figure describes a supplier process using Business Process Modeling Notation (BPMN). The supplier process contains two swimlanes, one for the sales unit and one for the invoicing unit. A process modeler wants to define the tasks A, D, and E in the sales swimlane as a process fragment, which represents the standard behavior of the sales unit. This process fragment can be reused in modeling new process models, in which the standard working process of the sales unit is needed.

The designed process fragment consists of two fragmental components: one fragmental component contains exactly one node, which represents the task A; the other fragmental component consists of two nodes representing the tasks *D*

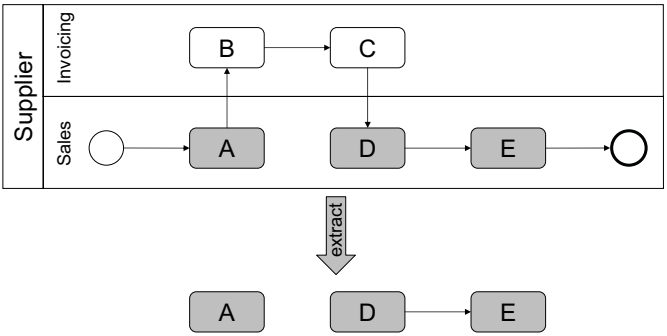


Fig. 3.3 A process fragment graph can also contain more than one weakly connected component. In a graph representation, the extracted process fragment contains two weakly connected components.

and E respectively and an edge that connects D and E. Each of the fragmental components represents a weakly connected graph.

Note that the control dependencies between the fragmental components are undefined. They can be determined at the time of reuse. For example, the process fragments shown in Figure 3.3 can be reused with a parallel gateway. It can be reused when creating a sequence flow that originates from task A and ends at task D. Allowing more than one fragmental component and weak connectedness between them provides process modelers more flexibility and increases the reusability of the process fragment.

3.4 Shapes of Process Fragments

The shape of a process fragment is defined in terms of the corresponding numbers of its entry and exit points. An entry point of a process fragment is either a start node or an incomplete incoming edge. A start node has no incoming edges. We use $d^{\leftarrow}(v)$ to indicate the number of the incoming edges of a node v, called **in-degree**, and $d^{\rightarrow}(v)$ to indicated the number of the outgoing edges of a node v, called **out-degree**.

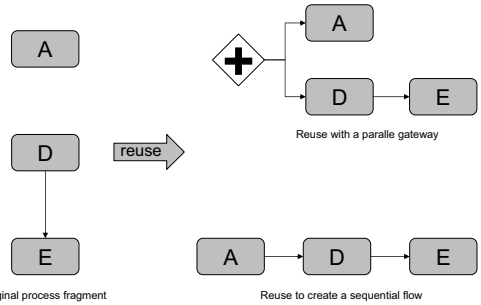


Fig. 3.4 The control dependencies of the fragmental components of a process fragment will be determined at the time of reuse. The original process fragment shown in the figure can either be reused with a parallel gateway to create a parallel process logic or be reused by creating a sequence flow that originates from task A and ends at task D.

A node v in a process fragment graph is a start node, if and only if its in-degree $d^{\leftarrow}(v) = 0$. Analogously, a node in a process fragment graph is an end node, if and only if its out-degree $d^{\rightarrow}(v) = 0$.

An incomplete incoming edge is an edge e with no start node but an end node, i.e. $\pi_1(e) = \perp$ and $\pi_2(e) = v$ with $v \in V$. Analogously, an incomplete outgoing edge is an edge e with a start node but no end node, i.e. $\pi_1(e) = v$ with $v \in V$ and $\pi_2(e) = \perp$.

We use $start(G')$ to denote the set of entry points and $end(G')$ the set of exit points of a process fragment graph G'. For a given process fragment graph $G' = (V', E')$ we define:

$$start(G') = \begin{cases} \{v'_1, ..., v'_m\} \cup \{e'_1, ..., e'_n\} & \text{if } d^{\leftarrow}(v'_i) = 0 \\ & \text{with } v'_i \in V' \text{ and } 1 \leq i \leq m; \text{ OR} \\ & \pi_1\left(e'_j\right) = \perp \text{ and} \\ & \pi_2\left(e'_j\right) = v' \text{ with } v' \in V' \; 1 \leq j \leq n \\ \emptyset & \text{otherwise} \end{cases} \tag{3.4.1}$$

$$end(G') = \begin{cases} \{v'_1,...,v'_s\} \cup \{e'_1,...,e'_t\} & \text{if } d^{\rightarrow}(v'_j) = 0 \\ & \text{with } v'_j \in V' \text{ and } 1 \leq i \leq s; \text{ OR} \\ & \pi_1\left(e'_j\right) = v' \text{ with } v' \in V' \text{ and} \\ & \pi_2\left(e'_j\right) = \perp\ 1 \leq j \leq t \\ \emptyset & \text{otherwise} \end{cases} \tag{3.4.2}$$

Based on the numbers of the entry and exit points of a process fragment we identified four different shapes of process fragments:

- **Single-Entry-Single-Exit (SESE):** a process fragment is in the shape of *Single-Entry-Single-Exit (SESE)*, if $|start(G')| = |end(G')| = 1$. We call such a process fragment a SESE process fragment.
- **Single-Entry-Multiple-Exits (SEME):** a process fragment is in the shape of *Single-Entry-Multiple-Exits (SEME)*, if $|start(G')| = 1$ and $|end(G')| > 1$. We call such a process fragment a SEME process fragment. The SEME process fragment shown in Figure 3.5 is a process fragment with Single-Entry-Two-Exits or a SE2E fragment for short.
- **Multiple-Entries-Single-Exit (MESE):** a process fragment is in the shape of *Multiple-Entries-Single-Exit (MESE)*, if $|start(G')| > 1$ and $|end(G')| = 1$. We call such a process fragment a MESE process fragment. The MESE process fragment shown in Figure 3.5 is a process fragment with Two-Entries-Single-Exit or a 2ESE fragment for short.
- **Multiple-Entry-Multiple-Exit (MEME):** a process fragment is in the shape of *Multiple-Entries-Multiple-Exits (MEME)*, if $|start(G')| > 1$ and $|end(G')| > 1$. We call such a process fragment a MEME process fragment. The MEME process fragment shown in Figure 3.5 is a process fragment with Three-Entries-Three-Exits or a 3E3E fragment for short.

We call a SESE process fragment a *simple process fragment*. A process fragment with single entry but multiple exit points is called a *fork process fragment*, because if we only consider the entry and the exit points of the process fragment the shape of the process fragment is like a fork . Analogously, a process fragment with multiple entry and single exit points is called a *join process frag-*

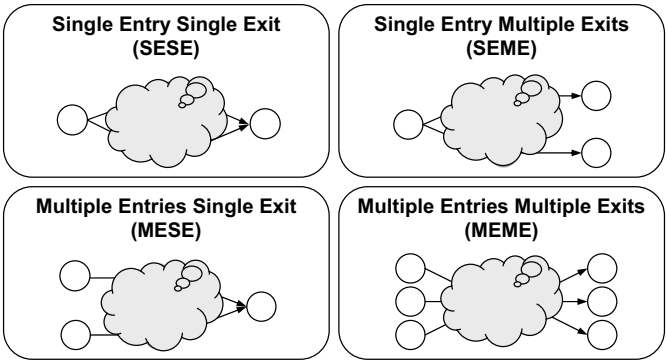


Fig. 3.5 The four shapes of process fragments are characterized by the corresponding numbers of the entry and exit points.

ment. A *complex process fragment* refers to a process fragment with multiple entries and multiple exit points.

The shape is one fundamental characteristic of process fragments that we have identified in the research. First, the shape information can be exploited by process modeling tools to make suggestions on how to stitch process fragments together [48]. Also, the shape is useful when using process fragments to refactor existing process models [147].

Second, the shape can be also used as a query criterion. For example, a process modeler may formulate the query request like: *give me all process fragments for risk assessment that has exactly one entry point and three exit points*. Considering the shape of process fragments in query processing is especially required when process modelers want to substitute a part of an existing process with a process fragment. When other parts of the original process model must not be modified, the process modeler prefers to find a process fragment whose shape fits exactly to the rest of the process model being designed.

3.5 Granularity of Process Fragments

Granularity is commonly used to describe to which extend a problem is broken down into smaller parts. Accordingly, the granularity of process fragments refers to which extent a process model can be broken down into smaller pieces for reuse. The concept of process fragments enables the reuse of one or more arbitrary parts of a process model, but does not impose restrictions on the syntactic and semantic completeness as (sub)processes do. Moreover, the granularity of a process fragment spans from an activity to the whole process.

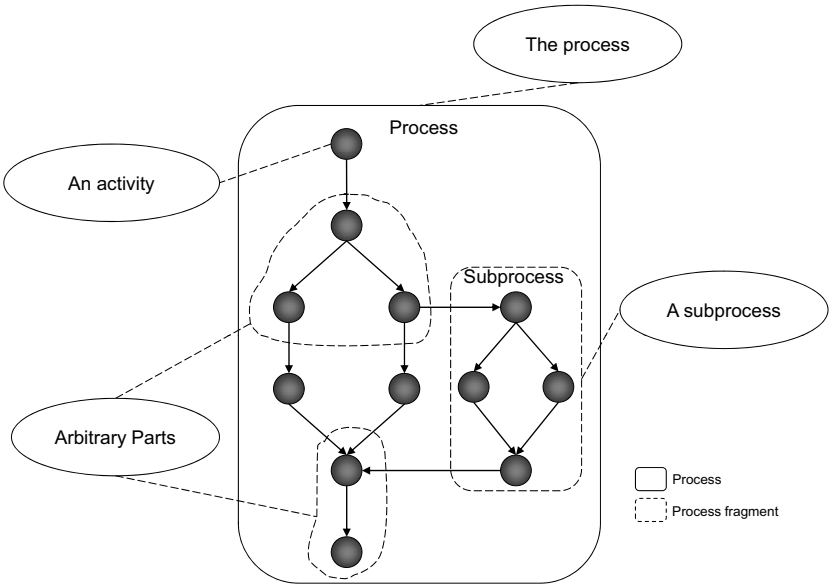


Fig. 3.6 Process fragments can encapsulate reusable process logic in different granularities.

The term *fragment* indicates a relative relation. A process model P may be used as a part to create a new process model. Thus, from the perspective of the new process model, the process model P can be considered as a reusable process fragment. A subprocess itself also represents a reuse granule of self-contained process logic. Processes and subprocesses can be decomposed fur-

ther into activities. An activity defines a piece of work that needs to be carried out as a part of a process. It is used as an atomic unit in modeling process logic. Thus, an activity represents also an atomic unit of modeling process fragments. For that reason, a process fragment must contain at least one activity. Besides the mentioned reuse granules, the concept of process fragment for reuse aims to enable the reuse of arbitrary parts of a process model. These arbitrary parts do not have to be syntactic and semantic complete, which is not allowed by activities, subprocesses and processes.

3.6 Reuse Styles of Process Fragments

Considering how process fragments are reused we have identified six typical reuse styles as shown in Figure 3.7. These reuse styles are characterized by the exposure and modifiability of the internal process logic of a process fragment.

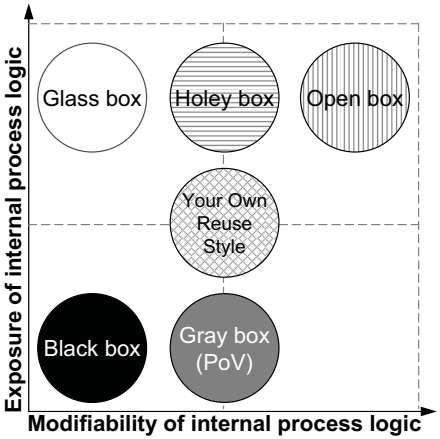


Fig. 3.7 Six typical reuse styles characterized by the exposure and modifiability of the internal process logic of a process fragment.

3.6.1 Black Box

Black box reuse style process fragments cannot be modified at the time of reuse and the internal definition of process logic is not exposed to process modelers reusing it.

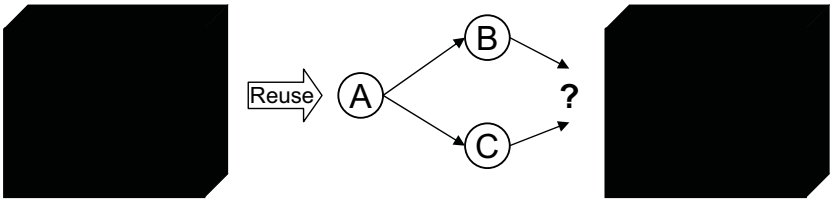


Fig. 3.8 Without the insight into the process fragment black box reuse style may confuse process modelers about how to connect the existing process part with the process fragment.

Process modelers have no access to modify or even view the internal process logic and implementation of the process fragment. Generic services, such as process model validation or data transformation, are usually reused as black boxes. Some commercial off-the-shelf software products also belong to this reuse style. The software product defines the input that it needs for processing and returns the results to process modelers. The procedures that it conducts in between and how the procedures have been implemented are totally invisible to process modelers.

The black box reuse style is difficult to apply to process fragments. When reusing process fragments process modelers have to ensure that the control flow and data flow of the resulting process model is correct. For that reason, they may need the insight into the process logic of the process fragment. Without the insight into the process fragment process modelers may be confused about how to connect the existing process part with the process fragment.

3.6.2 Gray Box

The gray box reuse style requires that the internal definition of process logic is not exposed to the fragment consumer. The fragment consumer can only affect the internal process logic by setting predefined parameters of the process fragment.

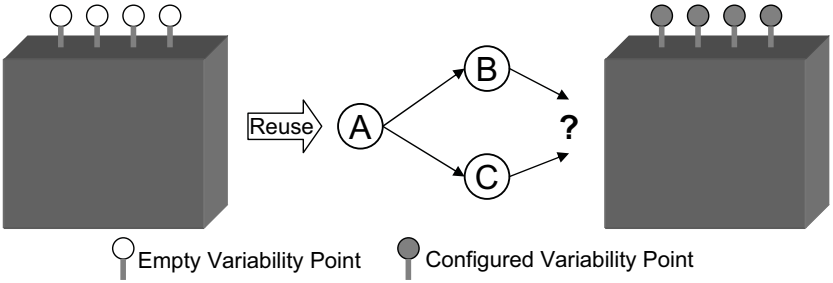


Fig. 3.9 Despite the predefined variability points process modelers are still kept in dark about the internal process logic, which hampers the integration of the process fragment with the process part being modeled.

Most of the commercial off-the-shelf components provide customers the possibility to parameterize the component to suit their individual needs. Such configurable parameters are called Points of Variability (PoV) or variability points. In addition, some services or components have deployment parameters that can be configured to suit diverse runtime environments. An example in the SOA world, is the configuration of the endpoints of Web services [155]. Parameterization has also be used to enhance the reusability of orchestration of Web services [75].

In comparison with black box reuse style, gray box reuse style provides more modifiability by allowing process modelers to configure the variability points. However, process modelers have no knowledge about the internal process logic, which leads to the same issue we have with black box reuse style. In other words, without the insight into the process fragment process modelers may still be confused about how to connect the existing process part with the process fragment.

3.6.3 Glass Box

The process fragment cannot be modified at the time of reuse, but the internal definition of process logic together with the process context are exposed to process modelers for evaluation or reviewing purpose.

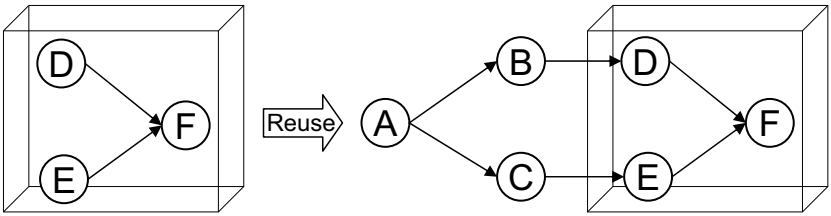


Fig. 3.10 Glass box reuse style provides insight into the process logic of the process fragment but does not allow process modelers to modify it.

As the name indicates, in the *glass box* reuse style a process modeler has the insight in the internal process logic of the process fragment, but has no access to modify it. Glass box reuse style can also be found in current process modeling scenarios. Companies may define subprocesses that prescribe standardized procedures to deal with a certain activity, such as invoice processing or risk assessment. This kind of subprocesses must be obeyed within the company without any deviation. When reusing such subprocesses, their internal process logic is visible to process modelers for transparency. However, process modelers are not allowed to modify it.

The glass box reuse style is suitable for process fragments that can be reused without any modifications. Process fragments that encode standardized and full-fledged process logic, e.g. compliance fragments [134], are appropriate candidates for the glass box reuse style.

3.6.4 Holey Box

The internal process logic is completely visible to process modelers and process modelers can only modify or customize the process logic at predefined positions.

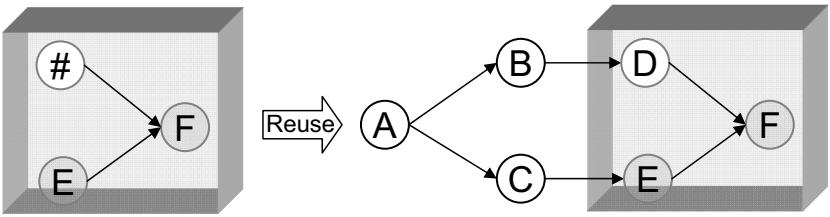


Fig. 3.11 Holey box reuse style allows process modelers to customized variability points. The variability point # in the process fragment on the left side has been replaced by the activity *D* in the process fragment on the right side.

Continuously changing business demand may quicly make a process fragment with rigid process logic obsolete. To increase and maintain the reusability of a process fragment process modelers may want to replace specific process activities or properties with placeholders, called variability points or parameters, so that they can be customized and completed when reusing the process fragments. The *holey box* reuse style enables process modelers to complete predefined variability points while keeping the rest of process logic untouched as in the glass box.

The *holey box* reuse style is especially useful for process fragments with variability points. Such process fragments can be reused as templates in process modeling. The variability points allow process modelers to customize and modify at predetermined positions, while the other parts of the process fragments ensure that the predefined process logic cannot be changed.

3.6.5 Open Box

The internal process logic is completely visible to process modelers and process modelers can modify or customize the process logic freely.

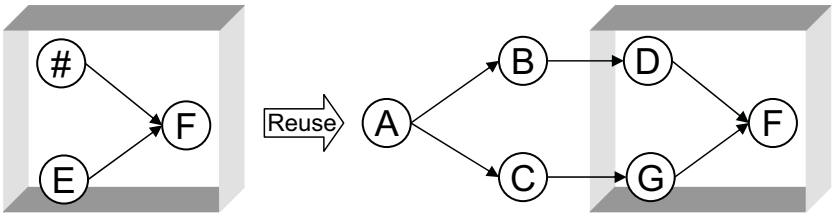


Fig. 3.12 Open box reuse style gives process modelers the most freedom on modifiability while ensuring the most visibility as in glass and holey box reuse styles. The variability point # in the process fragment on the left side has been replaced by the activity *D* in the process fragment on the right side. The activity *E* on the left side has been replaced by the activity *G* on the right side.

The *open box* reuse style provides process modelers the most visibility and modifiability of the process logic defined in a process fragment. It allows process modelers to add new process activities, delete obsolete process activities, modify existing process activities, and even change the control flow of the process logic. We can find the open box reuse styles in the current reuse approaches for process modeling. By using reference process models, so called best-practice process models, process modelers can extend the predefined process models with additional activities to incorporate company-specific business logic and refine the reference process models with implementations to make it executable [77]. Model-by-example allows process modelers to take an existing process model and to modify it respectively to make it meet current requirements. Both approaches provide process modelers with the insight into the internal structure and implementation of process models and allow process modelers to modify them when needed.

Despite its high flexibility in terms of modifiability the degree of freedom changing the process logic may destroy the original intent of the process fragments.

3.6.6 Customized Reuse Style

The reuse styles are not limited to those that we discussed above. Organizations may define their own customized reuse styles that combine the typical reuse styles.

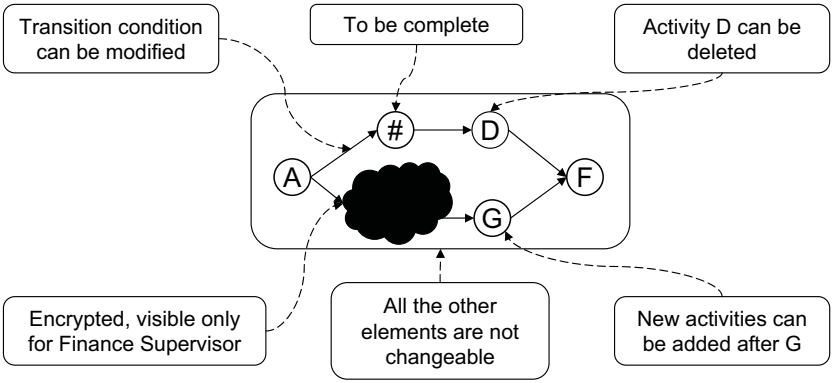


Fig. 3.13 An example of a process fragment with customized reuse style that combines the black box, glass box, holey box, and open box reuse styles.

Figure 3.13 shows a customized reuse style. This customized reuse style combines the black box, glass box, holey box, and open box reuse styles. As the process fragment in Figure 3.13 contains mission-critical process logic, which must not be visible to any process modelers, the organization applies the black box reuse style here to hide the internal implementation of this part of process logic. The holey reuse style is applied to allow only pre-defined placeholders to be completed at the time of reuse. The same is applied to the transition condition between the activity *A* and the placeholder activity. The activity *D* may not be always needed in all possible business context, for that reason, the open box reuse style allows process modelers to change the control flow by deleting the activity *D*. Analogously, new activities can be added after *G*. Besides the defined rules, all other activities like *A* and *F* must not be modified, but are visible to process modelers. Therefore, it applies the glass box reuse style for them.

3.7 Process Fragments Modeling Lifecycle

Process fragments modeling lifecycle refines the process design phase in the conventional business process management lifecycle [5]. The refined lifecycle provides a guideline to process modelers for a better understanding and application of the reuse concept in process modeling. On the other hand it can also guide the development of process modeling tools and process repositories for a better tooling support [99]. The process fragments modeling lifecycle consists of the following seven phases: identification, design, annotation, storage, retrieval, customization, and composition.

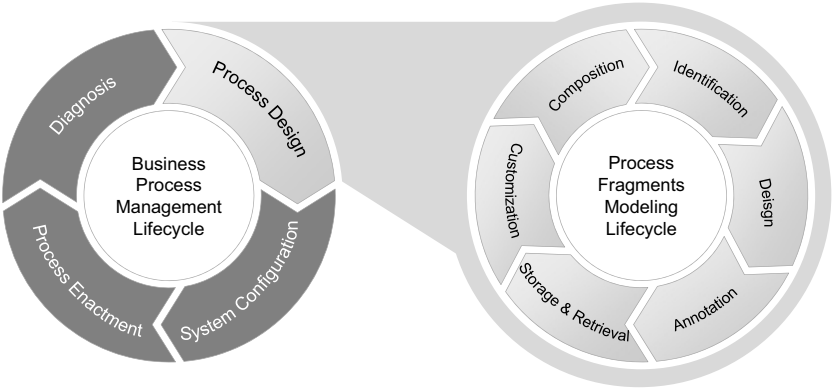


Fig. 3.14 Process fragment modeling lifecycle refines the process design phase in the conventional business process management lifecycle.

In this thesis, we address the design phase in Chapter 4 by introducing the BPEL fragment modeling language and in Chapter 5 by introducing a design method for extracting BPEL fragments from an existing BPEL process. As a proof of concept we provide a prototype of a BPEL process modeling tool, which extends Eclipse BPEL Process Designer and implements the extracting mechanism. For the storage and retrieval phases we provide a prototype of a BPEL repository for storing BPEL process models and fragments. Especially for the retrieval phase, we present in Chapter 7 an algorithm for querying BPEL process models and fragments that have approximate process structure. The

identification, annotation, customization, and composition phases are out of scope of this thesis. Related research works for these phases can be found in Chapter 2 related work.

3.7.1 Identification

The identification phase aims to determine candidate of process fragments, whose process logic represents high reusability in modeling new process models. Occurrence frequency is a useful metric for identifying reusable process fragments, which indicates the agnosticism of the partial process logic. The higher the occurrence frequency of the partial process logic, the higher the possibility that the process logic can be reused in modeling new process models. However, the definition of the lower bound of the occurrence frequency depends on each individual domain.

For identifying reusable process fragments the relations and dependencies between business processes within given business domains should be analyzed. The results may be used to confine the target scope for identifying process fragments, in order to avoid considering the whole business process landscape, which may lead to time-consuming work and high analysis complexity.

In order to detect candidates for reusable process fragments, the selected business processes should be compared with each other for repeated or similar subset of process logic. The comparison can be carried out in a manual or semi-automatic manner. In the manual manner, process modelers must go through each process and try to find frequent process fragments based on their comprehension of the process models. Despite the high requirement on process knowledge, this approach is also time-consuming and error-prone [84]. To provide process modelers the clue to potential candidates of frequent process fragments [62] presents a graph-based approach for mining frequent process fragments among a group of process models. In this semi-automatic approach process graphs are totally broken down to activity level. Users have to define the lower bound of occurrence frequency, based on which candidates of fre-

quent process fragments can be discovered. The identification phase is out of scope of this thesis.

3.7.2 Design

The candidate process fragments identified in the preceding phase may need to be extracted from the original process models. To extract a process fragment process modelers select the process activities that have been identified as constituents of a frequent process fragment. The modeling tool should help process modelers to extract identified process fragments. The original control dependencies between the activities of the resulting process fragment should be retained, if needed. Retaining the original control dependencies can ease the reuse, as process modelers do not have to figure out how the activities should be connected with each other.

The resulting process fragment may need to be re-designed. For example, process modelers may want to transform specific process activities or attributes into placeholders so that the process fragment can be reused as a template.

In case no frequent process fragments can be identified in the preceding phase, process fragments can be designed from scratch, especially when an organization is undertaking a process re-engineering project. Newly introduced process logic can be modeled as process fragments, if frequent applications of the process fragments can be reasonably predicted.

In Chapter 5 we introduce a method for extracting BPEL fragments from an existing BPEL process. As a proof of concept we provide a prototype of a BPEL process modeling tool, which extends Eclipse BPEL Process Designer and implements the extracting mechanism.

3.7.3 Annotation

One of the key issues in reusing process fragments is the ability to efficiently find the appropriate process fragments. The annotation phase gives process

modelers the possibility to enrich process fragments with metadata. Accurately defined and clearly documented vocabulary of metadata will provide a consistent means for annotating process fragments.

Some aspects of a process fragment are encoded in the process logic, such as functional capabilities and involved organizational units. Extracting these aspects of the process logic enables process modelers to search process fragments in a manner similar to keyword-based query [57]. Other aspects are not necessarily encoded in the process logic. The annotations allow process modelers to enrich the description of a process fragment with non-inherent properties, such as applied compliance rules, required performance metrics, etc. [159].

In the following we identified four fundamental and common aspects related to business processes annotations. Besides reuse styles and shapes, they represent only a subset of all possible metadata for a process fragment.

- **Business function** describes the functional capabilities of the process fragment, such as risk assessment, check room availability, request shipping schedule, etc.
- **Business object** defines the data that a process fragment processes, such as purchase order, invoice, etc.
- **Organizational unit** specifies the involved roles and organizational units in the process logic. For example, finance officer, production planer, teams, departments, partners, etc.
- **Policy** covers the non-functional properties of a process fragment, such as the performance metrics that the process fragment achieves, compliance rules that the process fragment follows, capability to compensate, etc.

An organization can extend these aspects to meet their individual requirements on the description of process fragments. Establishing an agreement on terms and usage between these descriptions and the business processes and constituent activities/transactions is a difficult but essential aspect of the overall methodology. The annotation phase is out of scope of this thesis.

3.7.4 Storage and Retrieval

The annotated process fragments should be stored and maintained in a consistent and systematic manner. A process repository is an established technology for managing engineering artifacts [30]. A process repository is a shared database of information about process modeling artifacts produced or used by an organization [30, 100, 115, 148]. It provides process modelers a central location to store and share reusable process modeling artifacts. In addition to conventional functionalities of a database a process repository also provides features such as checkout/check-in, versioning, configuration management, and access control.

When modeling a process, process modelers can query the process repository for process modeling artifacts that they can reuse. To increase the query capacity of the process repository and promote reuse of existing process modeling artifacts, the process repository should be able to return results that either exactly match the query request or represent approximate matches.

3.7.5 Customization

Exact or approximate matched process fragments may need to be customized before reuse to meet the needs of the requirements. For example, new process elements may be added; deprecated process elements may be deleted or replaced; the sequences of the process elements may be changed; process elements and attributes of process elements may need to be completed. The reuse styles (Section 3.6) determine which customization operations are allowed on which process elements of the process fragment. Tools should support the reuse styles and the needed techniques.

3.7.6 Composition

In the composition phase, process fragments can be stitched together [48] to specify more complex process logic. Process fragment composition can be conducted in a manual or (semi)automatic manner. In the manual manner, process modelers have to manually connect entry and exit points of process fragments. In the semi-automatic manner, the process modeling tool should suggest reasonable possibilities for composing the process fragments. The automatic manner allows the modeling tool to compose process fragments without human intervention. Both for the manual and the (semi)automatic manner, the shape of process fragments provides an important clue on how to stitch the process fragments together [48].

Chapter 4
Defining BPEL Fragments

This chapter introduces the BPEL fragment modeling language, which implements the concept of process fragments. In Section 4.2 we analyze the requirements on BPEL fragments and the language for modeling BPEL fragments. As BPEL provides two syntax variants for different purposes, we evaluate in Section 4.3 which syntax is the most suitable as the basis for BPEL fragment modeling language. In Section 4.4 we address the special needs for a new root element of BPEL fragments. As a result of the previous sections, we introduce in Section 4.5 the syntax of the BPEL fragment modeling language. Last but not least, not every arbitrary part represents a BPEL fragment according to the definition we introduced in this thesis. In Section 4.6 we compare BPEL fragments with such parts called BPEL modeling snippets.

4.1 Introduction

Web Services Business Process Execution Language (BPEL) [16] is the de jure standard for modeling both executable and abstract business processes. The current version of the BPEL language does not support the design of reusable parts of process logic, especially parts that cannot be seen as a self-contained process. The BPEL extension BPEL-SPE [82] was developed to foster reuse of BPEL

process logic by introducing subprocesses into BPEL: standalone subprocesses or inline subprocesses.

A standalone subprocess must be defined as a self-contained process, so that it can be invoked by another BPEL process. Thus, standalone subprocesses do not allow process modelers to design incomplete parts of a BPEL process for reuse.

An inline subprocess does not need to be a self-contained process, however, it can only be defined within a BPEL `<scope>`. Thus, inline subprocesses do not allow process modelers to reuse parts of a BPEL process within other modeling constructs, e.g. a `<flow>` activity.

As a conclusion, both standalone and inline subprocesses have restrictions on reusing parts of process logic in BPEL process models.

In this chapter we present one realization of the concept of process fragments which takes BPEL as the technological basis. In Section 4.2 we analyze the requirements on BPEL fragments. Some of the requirements are derived from the concept of process fragments themselves, while other requirements originate from the language design of BPEL itself. Based on the identified requirements we study in Section 4.4 whether one of the available BPEL constructs can serve as the root element of BPEL fragments. As none of the available BPEL constructs is suitable for this, we introduce in Section 4.5 the specification of the language for modeling BPEL fragments for reuse, the BPEL fragment modeling language (BPEL-Frag for short). It combines both the language constructs for modeling executable and abstract BPEL processes. Moreover, it enables process modelers to define arbitrary parts of a BPEL process as process fragments. Such arbitrary parts are not syntactically allowed by the BPEL language itself and its already defined extensions.

4.2 Requirements on BPEL Fragments

In this section we analyze the requirements on BPEL fragments. BPEL fragments must satisfy the requirements that are imposed by the definition of pro-

cess fragments (Definition 1). Specific requirements originate from the language design of BPEL itself.

Adhering to the BPEL Language (Requirement R1)

Generally speaking, there are two different approaches to design a BPEL fragment. One approach is to design a reusable BPEL fragment from scratch. Another approach is to design a BPEL fragment by reusing extracted parts of existing BPEL processes (see Section 3.7.2 in process fragment modeling lifecycle). The extracted parts of existing BPEL processes may contain BPEL constructs of either executable BPEL or abstract BPEL. As abstract BPEL allows the use of all BPEL constructs of executable BPEL, it is sufficient if the BPEL fragment modeling language provides constructs of abstract BPEL. In addition, BPEL fragment modeling language should relax the rigid constraints of the original BPEL constructs and provide new constructs, if needed, to meet the following requirements.

Requirement 1: the syntax of BPEL fragments should take BPEL as the common base and extend it to meet the requirements discussed in the following subsections.

Containing At Least One Basic Activity (Requirement R2)

A process fragment must contain at least one activity (Definition 1). As an implementation of process fragments a BPEL fragment must also contain at least on activity.

BPEL defines two different kinds of activities: basic and structured activities. Basic activities specify elemental steps of the business logic, which is used as atomic units for encoding the actual business logic. Structured activities are used to organize basic activities into a logical structure and to define their control dependencies, but do not describe a piece of work that forms a logical step within a process [3]. As the concept of BPEL fragments is introduced to en-

able process modelers to reuse existing process logic, a BPEL fragment must contain at least on basic activity.

Requirement 2: a BPEL fragment must contain at least one basic activity. The basic activity can either be located immediately within the root element of the BPEL fragment or be enclosed in any compound constructs in arbitrary depth.

Using Syntactically Incomplete Constructs (Requirement R3)

A process fragment may be syntactically incomplete (Definition 1). As we discussed in R1, the syntax of BPEL fragment modeling language should extend the common base of the BPEL language. But the syntactic constructs of executable BPEL language have mandatory constituents. For example, an `<if>` activity must have one mandatory *if* branch, which contains exactly one condition and exactly one activity. However, under certain circumstances a process modeler may only want to capture the default process logic in the `<else>` branch in a BPEL fragment. Thus, we have identified the following requirement:

Requirement 3: the syntax of BPEL fragment modeling language should allow process modelers to use syntactically incomplete constructs in modeling BPEL fragments. For that purpose, the syntax should eliminate the syntactical constraints on mandatory constituents of BPEL constructs, so that process modelers can leave out certain parts of the process logic when needed[1].

Using Variability Points (Requirement R4)

A process fragment may also be semantically incomplete (Definition 1). As discussed in the preceding chapter, one of the major indicators of semantic incompleteness is the presence of variability points. The BPEL language provides opaque activity as an explicit placeholder for exactly one executable BPEL activity. But process modelers may need to define a variability point in a BPEL

[1] A similar requirement led to the definition of abstract BPEL

fragment, which can be replaced either by exactly one BPEL activity or by a BPEL fragment.

For example, a process modeler may want to create a BPEL fragment that contains the standardized activities for claims processing. Each team can reuse this BPEL fragment to model its local claims handling processes based on the local economic situation, legislation, man power, and other factors that cannot be foreseen at the time of modeling. Important is that all the claims handling processes begin with the activity *receiveClaimRequest* followed by the activity *getCustomerInfo*. Depending on whether the claim requester is a business customer or a private customer each local team should be able to decide on its activities according to their current priority to each customer category. All the claim processing processes end with the activity *recordResult* and the activity *closeCase*. Each team can define the activities between *getCustomerInfo* and *recordResult* individually. In this case, the process modeler does not known the local process logic at the time of designing this BPEL fragment, which could be one single activity or more than one activity.

Requirement 4: besides `<opaqueActivity>` the syntax of BPEL fragment modeling language should allow process modelers to define explicit placeholders that can be replaced by a BPEL fragment.

Including or Excluding Process Context (Requirement R5)

Another major indicator of semantic incompleteness is the lack of adequate information about the process context (Definition 1). We use the term *process context* to refer the collection of variables, partner links, message exchanges, correlation sets, extension, and imported documents in a BPEL process.

Including process context helps process modelers in discovering and reusing BPEL fragments appropriately. This can be achieved by providing additional environmental information that goes beyond the pure process logic.

For example, a partner link specifies the relationship of the BPEL process and an interaction partner. Without the definitions of the partner links it is difficult to figure out through which communication channel an `<invoke>` activity interacts with the partner process. Similarly, variables carry the data that are

used in a BPEL process internally or exchanged between the BPEL process and its partners. In addition, when reusing a BPEL fragment as the basis to model a new BPEL process, including process context as part of the BPEL fragment avoids repeated modeling work.

Excluding the process context in a BPEL fragment should also be allowed. A BPEL fragment without process context consists of process logic only. Such a BPEL fragment allows process modelers to reuse it in different process contexts. At the time of reuse, process modelers can either define the process context from scratch or embed the BPEL fragment into an already defined process context.

Requirement 5: the syntax of BPEL fragment modeling language should allow process modelers either to include process context in or to omit process context from a BPEL fragment.

Defining a Single Activity as a BPEL Fragment (Requirement R6)

The BPEL language uses activities as modeling constructs to specify pure process logic. When decomposing a BPEL process into BPEL fragments, the most intuitive way is to split the BPEL process into different kinds of activities. Each of these obtained activities could represent a reusable BPEL fragment for process modelers.

For example, the obtained BPEL fragment could be a basic activity: because it is time-consuming and error-prone to define an `<assign>` activity manually, especially when it is dealing with complex data mapping procedures, reuse of such an assign activity is a valid use case for a single activity fragment. Thus, one process modeler may define a fully-specified `<assign>` activity as a BPEL fragment and many other process modelers may reuse it later in different BPEL processes when the same data mapping rules apply.

As another example, the obtained BPEL fragment could be a complex structured activity: a `<scope>` activity in an order-to-cash process encodes the standardized process logic for billing customers. In addition it defines also corresponding fault handlers for exception handling and compensation handler in case of rollback. As the process logic encoded in the `<scope>` is standard-

ized and can be reused across more than one order-to-cash processes, process modelers may also define the <scope> as a BPEL fragment.

Requirement 6: the syntax of BPEL fragment modeling language should allow process modelers to define a single BPEL activity as a BPEL fragment.

Delayed Decision on Control Dependencies (Requirement R7)

When designing reusable BPEL fragments, a process modeler may want to determine the control dependencies of the enclosed activities at the time of reuse.

For example, a process modeler wants to create a BPEL fragment that comprises two activities: *delivery orders* and *billing customer*. The process modeler intends to reuse the BPEL fragment in modeling new *order-to-cash* processes. The control dependencies of the activities *delivery orders* and *billing customer* depend on the actual business logic that should be described by reusing the BPEL fragment. Assume that a company has the following policies: (i) for key accounts these two activities should run in parallel in order to reduce the delivery time; (ii) for private customers the activity *delivery orders* can be started only if the activity *billing customer* has been completed successfully. In case (i), the two activities can be embedded in a <flow> activity; in case (ii), the two activities can be reused within a <sequence> activity in the proper order.

Note that control dependencies of a group of activities are either specified implicitly by the immediate nesting construct or explicitly using links. Thus, to enable delayed decision on control dependencies of the immediate child elements in a BPEL fragment, we identified the following requirement:

Requirement 7: the syntax of BPEL fragment modeling language should allow using BPEL constructs as its immediate child elements without an immediate enclosing construct and BPEL links.

Associating Partial Process Logic with Condition (Requirement R8)

In BPEL, the bundling of a <condition> element and an activity can be used in two cases: for specifying conditional behavior and for defining repetitive process logic.

The bundling of a <condition> element and an activity is used in <if> activities for specifying conditional behavior. Each branch of an <if> activity consists of a pair of a <condition> and an activity. The activity of the branch, whose condition has been first evaluated to be true, is performed.

The bundling of a <condition> element and an activity is also used in modeling <while> and <repeatUntil> activities. In these activities the <condition> controls the iteration of repetitive execution and the activity defines the actual process logic for each iteration.

As <if>, <while>, and <repeatUntil> activities share the same syntactical constituents, the bundling of a <condition> element and an activity can be considered as a granule of reuse, which allows process modelers to define building blocks for modeling conditional behavior and repetitive process logic.

Note that the BPEL language allows to associate exactly one activity with a <condition> element. To design a reusable BPEL fragment, a process modeler may want to associate a partial process logic, which may contain more than one activity with a condition. Let us consider the example discussed in the section of Requirement R7. Besides the delayed decision on the control dependencies of the activities *delivery orders* and *billing customer*, the process modeler wants to associate a condition with the partial process logic. The condition determines when the activities should be executed. But how the activities should be executed, i.e. in which execution order, will be determined by the delayed specification of the control dependencies between the two activities at the time of reuse.

Requirement R8: the syntax of BPEL fragment modeling language should allow to associate a <condition> element with one or more activities and use them as its immediate child elements.

Grouping of Handlers (Requirement R9)

A group of BPEL handlers alone could also represent a reusable part of a BPEL process.

For example, a process modeler may want to define the fault handlers *creditCardCanNotBeAuthroized*, *creditCardCanNotBeCharged*, and *creditCardIsInvalid*, and the compensation handler *cancelPurchageOrder* as a BPEL fragment. These handlers together represent a logical group of process logic for dealing with frequent exceptions that may be thrown during payment processing of credit cards. Thus, they could be identified as a reusable BPEL fragment.

BPEL uses the `<scope>` activity to bundle actual process logic with BPEL handlers. Using a `<scope>` activity for the purpose of grouping BPEL handlers would introduce redundant nesting of a `<scope>` activity in the target BPEL fragment.

Requirement 8: the syntax of BPEL fragment modeling language should allow to define a group of BPEL handlers without the nesting `<scope>` activity.

Associating Partial Process Logic with Handlers (Requirement R10)

Only grouping BPEL handlers is not sufficient for designing reusable parts of a `<scope>` activity. As handlers are generally used in combination with a `<scope>` activity, under certain circumstances they cannot be considered and reused in isolation. Thus, when designing a primary activity of a `<scope>` activity as a BPEL fragment, a process modeler may also want to include the corresponding BPEL handlers that are related to the partial process logic of the BPEL fragment being designed.

For example, a process modeler may want to create a BPEL fragment that comprises two activities, i.e. *delivery orders* and *billing customer* and decide on their control dependencies at the time of reuse (refer to the example used for Requirement R8). In addition, the process modeler wants also to include the fault handlers *creditCardCanNotBeAuthroized*, *creditCardCanNotBeCharged*, and *creditCardIsInvalid* to handle the exceptions that could be caused by the

activity *billing customer*. The original <scope> activity cannot be used to serve this purpose, as its syntax requires exactly one activity as its immediate primary activity.

Requirement R10: the syntax of BPEL fragment modeling language should allow associating partial process logic with BPEL handlers without the nesting <scope> activity.

Extensibility (Requirement R11)

The syntax of the BPEL fragment modeling language provides only the syntactical constructs for describing reusable process logic. The actual process logic of each BPEL fragment and the evaluation of its reusability depend on each individual application domain. Therefore, it is impossible to exhaust all the needs for different domains in this thesis. The identified requirements on BPEL fragment modeling language above represent the reasonable ones from the perspective of the author.

Requirement R11: the syntax of BPEL fragments should be extensible. The extensibility should allow process modelers to use the extended elements defined with the extension mechanisms provided by the original BPEL language. Also, the extensibility should allow process modelers to introduce new types of BPEL fragments that are not foreseen in the BPEL fragment modeling language.

4.3 Deciding on the Basis of BPEL Fragment Syntax

BPEL provides two language variants for specifying BPEL processes: one for modeling executable BPEL processes and one for specifying abstract BPEL processes. In this section we examine which language variant we can use as the basis for defining BPEL fragment modeling language (R1). The comparison of the two language variants is based on the identified requirements on BPEL fragments. Figure 4.1 shows the result of our comparison.

Requirements	Abstract BPEL Syntax	Executable BPEL Syntax
R1	●	○
R2	○	○
R3	●	○
R4	●	○
R5	●	●
R6	●	●
R7	○	○
R8	○	○
R9	○	○
R10	○	○
R11	○	○
	● Satisfies	○ Dissatisfies

Fig. 4.1 Comparison of the syntax of abstract and executable BPEL.

Instead of simply choosing the language variant (i.e. abstract BPEL) that satisfies most of our requirements, we argue in the following that the manner in which abstract BPEL satisfies the requirements makes it more suitable as the basis of the modeling language of BPEL fragments.

Expressiveness

Abstract BPEL provides more expressive power than executable BPEL. Abstract BPEL shares the common base of modeling constructs that are also allowed for executable BPEL. In addition, abstract BPEL introduces opaque constructs as explicit placeholders, which are not allowed in executable BPEL. Thus, abstract BPEL is preferable from the expressiveness perspective.

Syntactical Incompleteness

Abstract BPEL globally removes the cardinality constraints on the constituents of BPEL constructs: BPEL elements and attributes that are defined as mandatory in the syntax of executable BPEL processes are now defined as optional. This allows using BPEL constructs that are not syntactical complete with re-

gards to their definitions in executable BPEL (R3). Thus, from the perspective of syntactically incompleteness abstract BPEL is preferable.

Variability Points

Abstract BPEL provides opaque constructs for defining variability points, i.e. `<opaqueActivity>`, `<opaqueFrom/>`, opaque attributes, and opaque value `##opaque`. Defining variability points is not allowed in executable BPEL. Although the opaque constructs of abstract BPEL only partially satisfy the requirement on variability points R4, it is preferred over executable BPEL.

Design Decision

In summary, abstract BPEL surpass executable BPEL in the following aspects: (i) it provides more expressiveness; (ii) the opaque constructs can be used to define variability points in BPEL fragments; (iii) the elimination of cardinality constraints allows process modelers to use syntactical incomplete constructs to model BPEL fragments. Therefore, we choose abstract BPEL as the basis of the BPEL fragment modeling language.

4.4 The Need for a Separate BPEL Fragment Root

In this section we discuss whether we can reuse a BPEL construct as the root element for BPEL fragments. As we choose the syntax of abstract BPEL as the basis of BPEL fragment modeling language, the analysis takes only constructs defined in the syntax of abstract BPEL into consideration (see Figure 4.2). In the following we discuss only the entries in Figure 4.2 for `<scope>` and `<extensionActivity>` in detail. The detailed analysis of the other entries can be conducted analogously.

In the following discussion, we use the (+Rx) to indicate that the constructs satisfies the requirement x, while (-Rx) means the dissatisfaction of require-

ment x. As the requirement R1 is a requirement on the overall design of BPEL fragment modeling language, it cannot be evaluated on each single constructs. For that reason, we evaluate the requirement (R1) for each construct to "-" as a symbol for *undefined* in the Table 4.2.

The *Scope* Activity

The <process> construct is the root element of BPEL processes. Therefore, one may first consider whether the <process> construct can be reused as the root element of BPEL fragments. As the <process> construct is a special case of the <scope> activity, we discuss here only the <scope> activity. If the <scope> activity does not satisfy all the requirements, as a specialization of the <scope> activity, the <process> construct would also dissatisfy the requirements.

The <scope> activity in abstract BPEL allows process modelers to leave out constituents of BPEL constructs when needed (+R3). In addition, it allows using explicit placeholders, i.e. opaque activities, opaque attributes, opaque from, and opaque value, for defining variability points (+R4). The <scope> activity also allows defining local process context for its enclosed process activities (+R5). Process context are defined as optional in abstract BPEL. This allows process modelers to omit process context (+R5). The <scope> activity allows to use exactly one activity as its primary activity (+R6). Last but not least, in a <scope> activity process modelers can define Fault-, Compensation-, Termination- and Event-handlers (+R9).

However, the <scope> activity does not satisfy the following requirements: the <scope> activity allows maximal one primary activity as its immediate enclosed activity (-R7); <condition> is not allowed to be used immediately within the <scope> activity (-R8); as the <scope> activity does not satisfy R7 and R8, it also dissatisfy the requirement R10 (-R10); in BPEL the extensibility is introduced to define new activities and attributes, but the main structure of the scope cannot be changed (-R11).

The *Extension Activity*

The BPEL language defines extension mechanisms allowing process modelers to introduce new attributes or activities that are not defined in the BPEL standard. One extension mechanism allows introducing new attributes for any standard element. Another extension mechanism supports the definition of specific assign operations within an `<assign>` activity. Both of these extension mechanisms do not satisfy the identified requirements on BPEL fragments, thus, we do not discuss them here further.

The `<extensionActivity>` element represents another extension mechanism of BPEL, which allows process modelers to introduce new activity types. The content of an extension activity must be a single element qualified with a namespace that is different from the BPEL namespace [16].

This requirement does not allow to use a BPEL activity or any standard elements of BPEL as the single element within an *extension activity*, because the namespace of the single element must be different than the BPEL namespace (-R5, -R6). In addition, an extension activity requires a single element as its immediately enclosed activity. This dissatisfies the requirement that the root element should be able to enclose more than one activities as the immediate enclosed elements of a BPEL fragment (-R7, -R8, -R9, -R10).

We evaluate requirements R2, R3, and R4 to be undefined, as they are not defined explicitly in BPEL. The evaluation depends on the each individual definition of extension activities.

Other BPEL constructs do not satisfy the basic requirement of the process fragment concept, i.e. a process fragment must contain at least one activity. Thus, we do not show them in the Figure 4.2 to keep it short. As the analysis in the Figure 4.2 shows, no constructs in BPEL satisfy the requirements on the root element of BPEL fragments. Therefore, we introduce a new root element into the BPEL fragment modeling language.

Name	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11
<assign>	●	○	●	●	○	○	○	○	○	○	○
<catch>	●	○	●	●	○	●	○	○	○	○	○
<catchAll>	●	○	●	●	○	●	○	○	○	○	○
<compensate>	●	○	●	●	○	○	○	○	○	○	○
<compensateScope>	●	○	●	●	○	○	○	○	○	○	○
<compensationHandler>	●	○	●	●	○	○	○	○	○	○	○
<empty>	●	○	●	●	○	○	○	○	○	○	○
<eventHandlers>	●	○	●	●	○	○	○	○	○	○	○
<exit>	●	○	●	●	○	○	○	○	○	○	○
<extensionActivity>	●	-	-	-	-	-	-	-	-	-	-
<faultHandlers>	●	○	●	●	○	○	○	○	○	○	○
<flow>	●	○	●	●	○	●	●	○	○	○	○
<forEach>	●	○	●	●	○	●	○	●	○	○	○
<if>	●	○	●	●	○	●	○	○	○	○	○
<invoke>	●	○	●	●	○	○	○	○	○	○	○
<onAlarmEvent>	●	○	●	●	○	●	○	○	○	○	○
<onAlarmPick>	●	○	●	●	○	●	○	○	○	○	○
<onEvent>	●	○	●	●	○	●	○	○	○	○	○
<onMessage>	●	○	●	●	○	●	○	○	○	○	○
<opaqueActivity>	●	○	●	●	○	○	○	○	○	○	○
<pick>	●	○	●	●	○	○	○	○	○	○	○
<process>	●	○	●	●	●	●	○	○	○	○	○
<receive>	●	○	●	●	○	○	○	○	○	○	○
<repeatUntil>	●	○	●	●	○	●	○	●	○	○	○
<reply>	●	○	●	●	○	○	○	○	○	○	○
<rethrow>	●	○	●	●	○	○	○	○	○	○	○
<scope>	●	○	●	●	●	●	○	○	○	○	○
<sequence>	●	○	●	●	○	●	○	○	○	○	○
<terminationHandler>	●	○	●	●	○	○	○	○	○	○	○
<throw>	●	○	●	●	○	○	○	○	○	○	○
<validate>	●	○	●	●	○	○	○	○	○	○	○
<wait>	●	○	●	●	○	○	○	○	○	○	○
<while>	●	○	●	●	○	●	○	●	○	○	○

● Satisfies ○ Dissatisfies - undefined

Fig. 4.2 No constructs of abstract BPEL in the table above satisfy all the identified requirements on BPEL fragments.

4.5 The BPEL Fragment Modeling Language

Based on the discussions above, it is clear that we do not only need to introduce new types of modeling constructs but also need to change the semantics of the root element of abstract BPEL and the allowed combinations of origi-

nal BPEL constructs. Therefore, instead of extending abstract BPEL by using the extension mechanisms provided by the language itself, we introduce the BPEL fragment modeling language as a new language variant into the BPEL language family. In the BPEL fragment modeling language the root element is the <fragment> element (4.5.2). In addition, the BPEL fragment modeling language introduces two new elements: (i) the <bagActivity> construct (4.5.1) is an extension of <opaqueActivity> and can be completed with one or more activities; (ii) the <extensionFragment> (4.5.2.4) which enables process modelers to introduce new types of BPEL fragments. Besides these changes, all the other constructs in the BPEL fragment modeling language are the same as those defined in abstract BPEL.

In general we will use the grammar representation for the discussions of BPEL fragment modeling language. However, under certain circumstances using XSD schema helps to make the discussion more clear and precise. Thus, in the following discussions XSD schema is used if necessary.

4.5.1 The Bag Activity

We introduce a new activity type called <bagActivity>, which has similar semantics as <opaqueActivity> of abstract BPEL. A <bagActivity> enables process modelers to use it as an additional explicit placeholder to model unknown process logic. It inherits the standard attributes and elements of all BPEL activities and is defined as follows:

```
<bagActivity standard-attributes>
  standard-elements
</bagActivity>
```

Listing 4.1 The structure of a bag activity.

The difference between the bag activity and the opaque activity is twofold: (i) a bag activity can be substituted by one or more abstract or executable BPEL activities, while an opaque activity represents an explicit placeholder for exactly one executable BPEL activity; (ii) a bag activity is allowed to have both

unified and ununified links (unified Links and non-unified Links are described in the following paragraph) while an opaque activity, as other BPEL activities, may only use unified links.

Substitution

One exemplary usage of bag activity is in the creation of BPEL fragments. A bag activity marks the points of variability in a BPEL fragment and indicates the unknown process logic at the time of design. A bag activity can be replaced by an executable or an abstract BPEL activity. It can even be replaced by another BPEL fragment.

Although a bag activity can be replaced by one or more BPEL activities, the immediately enclosing construct of the bag activity determines per definition how many BPEL activities are allowed to replace the bag activity. Some BPEL constructs require exactly one BPEL activity as their immediate enclosed activity, e.g. `<if>`, `<scope>`, `<catch>`, etc. A bag activity that is used as the immediate enclosed activity in such constructs can only be replaced by exactly one BPEL activity. Some other BPEL activities, i.e. `<flow>` and `<sequence>`, allow more than one BPEL activities as their immediate child activities. Thus, a bag activity that is used as an immediate enclosed activity of these BPEL activities can be replaced by one or more BPEL activities.

Unified Links

A bag activity indicates that the process logic is unknown at the time of modeling. If a bag activity has exactly one incoming link, the semantic of the single incoming link may be twofold: (i) the substitution must have exactly one entry point; (ii) the process modeler does not know what the process logic could be. Thus, she/he uses a single incoming link just as a means to connect the bag activity with other parts of the BPEL fragment. The same also applies to the single outgoing link of a bag activity. In order to distinguish the different

semantics of the unified incoming or outgoing link of the bag activity we introduce two different modes: the restricted mode and the relaxed mode.

The restricted mode is introduced for the former case (i). In the restricted mode, the unified incoming or outgoing link of the bag activity must not be modified when substituting the bag activity. This requires that the substitution of the bag activity must have exactly one entry point if the bag activity has exactly one incoming link. Analogously, the replacement must have exactly one exit point if the bag activity has exactly one outgoing link. The entry point of the substitution inherits the join condition of the bag activity. And the exit point of the replacement inherits the transition condition of the bag activity.

The relaxed mode enables the latter case (ii). In the relaxed mode, both the unified incoming and outgoing links can be replaced by a new set of incoming and outgoing links when substituting the bag activity. The join condition and transition condition of the bag activity should be verified after the substitution.

For example, the unified incoming link of a bag activity can be duplicated when substituting the bag activity with a BPEL fragment with more than one entry points. If the original incoming or outgoing link carries a transition condition, it can be copied to each link. However, the respective transition conditions should be verified whether they are still valid for the resulting process logic. If the target activity of the single outgoing link carries a join condition, then the join condition should be adjusted, if necessary, by process modelers to include also the duplicated links.

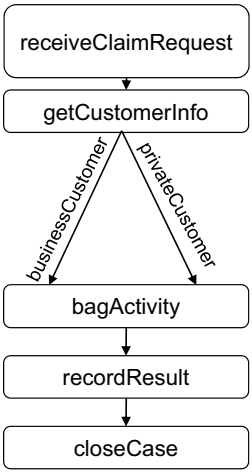
The mode should be specified in the BPEL fragment profile[2], which describes how to use the respective BPEL fragment, e.g. reuse styles applied, how to resolve unified links, etc. However, BPEL fragment profile is out of the scope of this thesis. We consider it as our future work.

Non-unified Links

When using a bag activity in a `<flow>` activity, the bag activity may have more than one incoming or outgoing links that all share the same source activity.

[2] similar to abstract BPEL profile

For example, a company has standardized some processing steps of their claim handling processes as shown in Figure 4.3. All the claim handling teams must follow these standardized steps. Each local team can define different activities for handling different customer categories accordingly: business customers and private customers. However, how they interact with their customers depends on local culture, legislation, economic situation, etc. Thus, they should be granted certain flexibility in designing the business processes. To accommodate this need, a process modeler creates a BPEL fragment for reuse, which contains the standardized steps and can be reused when modeling each local claims handling process. As the local activities must distinguish business customers from private customers, the bag activity has two incoming links with the corresponding transition conditions.



a) Use bag activity

Fig. 4.3 A bag activity may have non-unified incoming or outgoing links.

Such incoming links of the bag activity as shown in Figure 4.3 are called in this thesis non-unified links. Non-unified links violate the static constraint defined in the BPEL standard. The static constraint [SA00067] specifies that:

two different links MUST NOT share the same source and target activities; that is, at most one link may be used to connect two activities.

To solve the problem, we have three alternatives: (i) introduce a new link type into BPEL fragment modeling language; (ii) create for each incoming link a bag activity; (iii) relax the static constraint in BPEL fragment modeling language to enable this use case.

(i) Introducing new types of modeling constructs in the BPEL fragment modeling language leads to the need to transform specific modeling constructs in the BPEL fragment modeling language to the original BPEL constructs. The design of the BPEL fragment modeling language aims to reuse existing BPEL constructs as much as possible, in order to ease the reuse of BPEL fragments in modeling new BPEL processes. Thus, we do not follow this alternative towards a solution.

(ii) Instead of introducing new type of link we could split the bag activity to avoid non-unified links. For each new created bag activity through splitting we have to duplicate the outgoing links of the original bag activity. The splitting introduces both the redundancy of the bag activity and the redundancy of the outgoing links, which is against the reusability that we want to achieve with this work. In addition, the redundancy could make the substitution of the bag activities more difficult and error-prone. Thus, we do not prefer this alternative.

Based on this discussion we propose to use the alternative (iii) and relax the static constraint as follows: *a bag activity MAY have more than one incoming link that share the same source activity and MAY have more than one outgoing link that share the same target activity. When substituting the bag activity with standard BPEL constructs, all the non-unified links of the original bag activity must be resolved.*

4.5.1.1 The Namespace

In order to distinguish BPEL fragments from executable and abstract BPEL processes, we introduce a new namespace. The syntax of BPEL fragments is denoted under the following namespace:

http://www.iaas.uni-stuttgart.de/wsbpel/2.0/process/fragment

4.5.2 The Root Element

The construct <fragment> is the root element of a BPEL fragment. It allows process modelers to define the process context information, i.e. declaration of extensions, imported documents, partner links, variables, message exchanges, and correlation sets. These elements are defined as optional in the syntax, which enables process modelers either to include or exclude the process context information in BPEL fragments (R5). In addition, the syntax of BPEL fragments allows process modelers to design different types of BPEL fragments. In the following subsection we discuss each of these types and the newly introduced elements for modeling BPEL fragments in detail. We use ellipsis to denote the omission of the standard attributes of the root element and the process context in the syntax.

The static constraints discussed in this section are only valid for BPEL fragments. When reusing a BPEL fragment to construct a BPEL process, the syntax of the corresponding BPEL language variant and the corresponding static constraints must be followed.

The following listing shows the basic structure of a BPEL fragment.

```
<xsd:complexType name="tFragment">
   <xsd:complexContent>
     <xsd:extension base="tExtensibleElements">
       <xsd:sequence>
         <xsd:element ref="extensions" minOccurs="0"/>
         <xsd:element ref="import" minOccurs="0"
             maxOccurs="unbounded"/>
         <xsd:element ref="partnerLinks" minOccurs="0"/>
         <xsd:element ref="messageExchanges" minOccurs="0"/>
         <xsd:element ref="variables" minOccurs="0"/>
         <xsd:element ref="correlationSets" minOccurs="0"/>
         <xsd:element ref="faultHandlers" minOccurs="0"/>
         <xsd:element ref="compensationHandler" minOccurs="0"/>
         <xsd:element ref="terminationHandler" minOccurs="0"/>
         <xsd:element ref="eventHandlers" minOccurs="0"/>
         <xsd:element ref="condition" minOccurs="0"/>
         <xsd:element ref="links" minOccurs="0"
             maxOccurs="unbounded"/>
```

```
        <xsd:group ref="activity" minOccurs="0"/>
        <xsd:element ref="extensionFragment" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:NCName"
          use="optional"/>
      <xsd:attribute name="targetNamespace"
          type="xsd-derived:anyURI" use="optional"/>
      <xsd:attribute name="queryLanguage" type="xsd-derived:anyURI"
            default="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0"/>
      <xsd:attribute name="expressionLanguage"
          type="xsd-derived:anyURI"
            default="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0"/>
      <xsd:attribute name="suppressJoinFailure" type="tBoolean"
          default="no"/>
      <xsd:attribute name="exitOnStandardFault" type="tBoolean"
          default="no"/>
      <xsd:attribute name="abstractProcessProfile" type="xsd:anyURI"
          use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Listing 4.2 The Basic Structure of a BPEL Fragment

The <fragment> construct allows a graph-based modeling approach. Different from the <flow> activity, the <fragment> represents only a wrapper of the process logic in a BPEL fragment. It does not impose any semantics on the control dependencies of its immediately enclosed activities.

The group *activity* contains all the activities defined in BPEL. In addition, it contains also the new activity <bagActivity>. The attribute minOccurs specifies that the number of activities immediately enclosed within the root element is unbounded. The definition of process fragments requires that a process fragment must contain at least one activity. Taking this constraint in the schema definition into consideration would make the schema unreadable. Tools that implement the schema of BPEL fragment should conduct static analysis to make sure that the BPEL fragment being designed has at least one activity.

4.5.2.1 The Top-Level Attributes

The root element of a BPEL fragment is the `<fragment>` construct. It inherits the standard top-level attributes defined in abstract BPEL as shown in the Listing 4.2. The purpose of the inheritance is twofold: (i) the top-level attributes provides useful information for human beings and machines to better understand the process fragment; (ii) it saves repeated modeling work when process modelers use the BPEL fragment as the starting point for modeling a complete BPEL process.

We reuse the attribute `abstractProcessProfile` for process modelers to specify the profile for the BPEL fragment being designed. Here we followed the same design principle of reusing existing BPEL constructs as much as possible. The URI of the attribute value makes the profiles distinguishable. We recommend using the target namespace of BPEL fragment as the prefix of the URI. For example, a process modeler may associate a profile for using the BPEL fragment as a template. Then the URI could look like:

http://www.iaas.uni-stuttgart.de/wsbpel/2.0/process/fragment/simple-template/2012/03

4.5.2.2 Flow Fragment

A flow fragment contains one or more activities and optionally links that specify the control dependencies of the enclosed activities. The grammar of a flow fragment is defined as follows:

```
<fragment ...>
  ...
  <links>?
    <link name="NCName" />*
  </links>
  activity+
</fragment>
```

Listing 4.3 The structure of a flow fragment.

In case no links are used in a flow fragment, the control dependencies of the enclosed activities depend on the target construct in which the fragment content is reused. If the target construct is a <sequence> activity, then the enclosed activities should be executed in the same order as they have been specified. If the target construct is a <flow> activity, then all the enclosed activities should be executed in parallel. Note that a flow fragment does not necessarily origin from a <flow> activity. For example, we can define a subset of the enclosed activities in a <sequence> as a flow fragment.

Recalled that from the graph-based point of view a connected component of a BPEL fragment is called a fragment component. In case at least one link is defined in a flow fragment, then a fragment component that uses BPEL <link>s to explicitly define control flow can only be reused immediately within a <flow> activity. Activities that have neither incoming nor outgoing links can be reused within a block-structured or a graph-based modeling construct.

BPEL requires that *every link declared within a* <flow> *activity MUST have exactly one activity within the* <flow> *as its source* ***AND*** *exactly one activity within the* <flow> *as its target*. But a flow fragment may contain links that do not have a source activity or a target activity. We call such links *incomplete links*. An incomplete link is defined as follows:

Definition 3. Incomplete Link
An *incomplete link* is a link that has either no source activity or no target activity. An incomplete link that has no source activity is called *incomplete incoming link*; while an incomplete link that has no target activity is called *incomplete outgoing link*. □

An example of a BPEL fragment that contains incomplete links is illustrated in Figure 4.4. To enable the use of incomplete links in the BPEL fragment modeling language, we choose to relax the static constraint instead of introducing a new type of links. Introducing incomplete link as a new type of links would lead to the need of a transformation of the incomplete links to the original BPEL links when reusing a BPEL fragment with incomplete links to model a new BPEL process. The design of the BPEL fragment modeling language

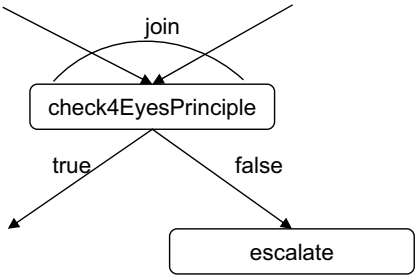


Fig. 4.4 An example BPEL fragment that contains incomplete links.

aims to reuse as much as possible the original BPEL constructs. Thus, we define the static constraint for the BPEL fragment modeling language as follows: *every link declared within a* `<fragment>` *elemement or a* `<flow>` *activity MUST have either exactly one activity within the* `<fragment>` *elemement or the* `<flow>` *as its source* ***OR*** *exactly one activity within the* `<fragment>` *element or the* `<flow>` *activity as its target.*

An incomplete link has an open end. Process modelers can either attach exactly one activity or a BPEL fragment to close the open end of the link. An incomplete link that has no source activity is equivalent to a BPEL link with a bag activity as its source. Similarly, an incomplete link that has no target activity is equivalent to a BPEL link with a bag activity as its target activity. Thus, we consider an incomplete link as a shortcut of a bag activity. We will discuss the usage of incomplete links in Section 5.2.3.2.

4.5.2.3 Conditional Fragment

A conditional fragment comprises of a `<condition>` element and one or more activities (R8). The grammar of a conditional fragment is defined as follows:

```
<fragment ...>
  ...
  <condition expressionLanguage="anyURI"?>bool-expr</condition>
  <links>?
```

```
    <link name="NCName" />*
  </links>
  activity+
</fragment>
```

Listing 4.4 The structure of a conditional fragment.

Constructs such as <if>, <while>, and <repeatUntil> share the same syntactical constituents, i.e. a <condition> element and an activity. Thus, conditional fragments can be used as building blocks for modeling conditional or repetitive behavior of a process.

4.5.2.4 Extension Fragment

As we discussed in the previous chapter, the identification of reusable BPEL fragments depends strongly on the individual business domain and its corresponding reuse scenarios. Thus, it is impossible to address all the needs of different domains in this thesis. Therefore, we introduce an extension mechanism into the BPEL fragment language (a similar mechanism as in BPEL) (R11), called *extension fragment*. The schema of the extension fragment is defined as follows:

```
<xsd:element name="extensionFragment" type="tExtensionFragment"/>
<xsd:complexType name="tExtensionFragment">
  <xsd:sequence>
    <xsd:any namespace="##other", processContents="lax"/>
  </xsd:sequence>
</xsd:complexType>
```

Listing 4.5 The XSD schema of extension fragment

The definition of an extension fragment enables process modelers to use any elements that are not defined in the target namespace of the BPEL fragments to define new types of BPEL fragments.

For example, a process modeler may use the BPEL extension BPEL4People [11] to model human workflows. After analyzing the process model the process modeler may want to define a BPEL fragment that contains a people activity and the definition of the potential owners. The potential owners are the person

who can claim and complete the human task [11]. As BPEL fragment modeling language is not capable of modeling human tasks, process modelers can use the extension mechanism to extend the expressiveness. Listing 4.6 shows an example.

```
<xsd:schema
    targetNamespace="http://www.iaas.uni-stuttgart.de/wsbpel/2.0/process/fragment"
     xmlns="http://www.iaas.uni- stuttgart .de/wsbpel /2.0/ process /fragment"
     xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/ abstract "
     xmlns:b4p="http://www.example.org/BPEL4People"
     xmlns:htd="http://www.example.org/WS-HT"
     ...>

...

<xsd:element name="peopleActivityWithOwners"
    type="tPeopleActivityWithOwners"/>
<xsd:complexType name="tPeopleActivityWithOwners">
  <xsd:complexContent>
    <xsd:extension base="tExtensionFragment">
       <xsd:element name="potentialOwners" type="htd:tPotentialOwners"
           minOccurs="0"/>
       <xsd:element name="peopleActivity" type="b4p:tPeopleActivity"
           minOccurs="0"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Listing 4.6 An example of extension fragment.

4.5.2.5 Broken Scope

When defining partial process logic of a <scope> activity as a BPEL fragment, process modelers may also enclose the corresponding handlers that are related to the activities used in the target BPEL fragment (R10). Therefore, we introduce a new type of BPEL fragment, called *broken scope*. A broken scope allows process modelers to associate BPEL handlers with partial process logic

of a scope, i.e. a flow fragment, a conditional fragment, or an extension fragment. The grammar of a broken scope is defined as follows:

```
<fragment ...>
  ...
  <compensationHandler>?
    activity?
  </compensationHandler>

  <faultHandlers>?
    <catch faultName="QName"?
      faultVariable="BPELVariableName"?
      ( faultMessageType="QName" | faultElement="QName" )? >*
      activity?
    </catch>
    <catchAll>?
      activity
    </catchAll>
  </faultHandlers>

  <terminationHandler>?
    activity?
  </terminationHandler>

  <eventHandlers>?
    <onEvent partnerLink="NCName"
      portType="QName"?
      operation="NCName"
      ( messageType="QName" | element="QName" )?
      variable="BPELVariableName"?
      messageExchange="NCName"?>*
      <correlations>?
        <correlation set="NCName" initiate="yes|join|no"? />+
      </correlations>
      <fromParts>?
        <fromPart part="NCName" toVariable="BPELVariableName" />+
      </fromParts>
      <scope ...>...</scope>?
    </onEvent>
    <onAlarm>*
```

```
    (
     <for expressionLanguage="anyURI"?>duration-expr</for>
     |
     <until
         expressionLanguage="anyURI"?>deadline-expr</until>
    )?
    <repeatEvery expressionLanguage="anyURI"?>?
     duration-expr
    </repeatEvery>
    <scope ...>...</scope>?
   </onAlarm>
  </eventHandlers>

  partialProcessLogic

</fragment>
```

Listing 4.7 The grammar of a broken scope.

The term *partialProcessLogic* in the Listing 4.7 represents a placeholder for a flow fragment, a conditional fragment, or an extension fragment. A broken scope with defined BPEL handlers can only be reused within the `<process>` construct or a `<scope>` activity. In this case, the partial process logic should be embedded into the primary child activity of the target scope properly and the handlers of the broken scope should be merged with the handlers of the target scope.

Note that merged fault handlers may have identical `<catch>` constructs. For BPEL fragments two `<catch>` constructs are considered to be identical, if their *faultName*, *faultElement* and *faultMessageType* attributes have identical non-opaque values. If an attribute is not present in a `<catch>` construct, its value is considered as opaque. Identical `<catch>` constructs should be resolved at the time of reuse.

4.5.2.6 Handler Fragment

If a BPEL fragment contains only fault, compensation, termination, or event handlers, then we call it a handler fragment (R9). It allows process modelers to

define a group of handlers that can be reused as a whole. The only difference between a broken scope and a handler fragment is that a broken scope contains partial process logic while a handler fragment does not. The grammar of a handler fragment is defined as follows:

```
<fragment ...>
  ...
  <compensationHandler>?
    activity?
  </compensationHandler>

  <faultHandlers>?
    <catch faultName="QName"?
      faultVariable="BPELVariableName"?
      ( faultMessageType="QName" | faultElement="QName" )? >*
      activity?
    </catch>
    <catchAll>?
      activity
    </catchAll>
  </faultHandlers>

  <terminationHandler>?
    activity?
  </terminationHandler>

  <eventHandlers>?
    <onEvent partnerLink="NCName"
      portType="QName"?
      operation="NCName"
      ( messageType="QName" | element="QName" )?
      variable="BPELVariableName"?
      messageExchange="NCName"?>*
      <correlations>?
        <correlation set="NCName" initiate="yes|join|no"? />+
      </correlations>
      <fromParts>?
        <fromPart part="NCName" toVariable="BPELVariableName" />+
      </fromParts>
      <scope ...>...</scope>?
```

```
    </onEvent>
    <onAlarm>*
      (
        <for expressionLanguage="anyURI"?>duration-expr</for>
        |
        <until
            expressionLanguage="anyURI"?>deadline-expr</until>
      )?
      <repeatEvery expressionLanguage="anyURI"?>?
        duration-expr
      </repeatEvery>
      <scope ...>...</scope>?
    </onAlarm>
  </eventHandlers>
</fragment>
```

Listing 4.8 The structure of a handler fragment.

As the cardinality indicates, fault handlers, compensation handler, termination handler, and event-handlers do not have to appear together in a handler fragment. A process modeler can define a single handler or any combination of the four handlers as a BPEL fragment. For example, a process modeler may have noticed that a scope for processing purchase order has always a compensation handler for cancellation the purchase order and a fault handler to deal with the case that the purchase order cannot be completed. In this case the process modeler can define a handler fragment that contains a compensation handler *cancelPurchaseOrder* and a fault handler *orderCanNotBeCompleted* as shown in the following list:

```
<fragment>
  <compensationHandler>
    <invoke partnerLink="Seller"
      portType="lns:purchaseOrderPT"
      operation="CancelPurchase"
      inputVariable="getResponse"
      outputVariable="getConfirmation" />
  </compensationHandler>
  <faultHandlers>
    <catch faultName="lns:orderCanNotBeCompleted"
```

```
    faultVariable="POFault"
    faultMessageType="lns:orderFaultType">
    <reply partnerLink="purchasing"
      portType="lns:purchaseOrderPT"
      operation="sendPurchaseOrder" variable="POFault"
      faultName="orderCanNotBeCompleted" />
  </catch>
 </faultHandlers>
</fragment>
```

Listing 4.9 An example of handler fragments.

4.5.3 Static Syntactical Constraint

A BPEL fragment must have at least one basic activity (R2). However, we do not encode this constraint into the syntax definition of BPEL fragment modeling language, as it would lead to verbosity and reduces its readability. Instead we require that the modeling tool should verify whether the modeled BPEL fragment satisfies this static constraint.

4.6 BPEL Modeling Snippets

A part of a BPEL process that does not contain a basic activity may also be reusable. As such reusable parts do not encode process knowledge, we call them reusable BPEL modeling snippets in order to distinguish them from BPEL fragments. In general, a reusable BPEL modeling snippet can be a contextual snippet or a structural snippet.

A contextual snippet contains process context such as variable, partner link, message exchanges, correlation sets, extension, imported documents, etc. A contextual snippet can be reused to complete the context information of a BPEL fragment. It avoids repeated modeling work and enables a rapid switching of process context for the same BPEL fragment.

A structural snippet contains no basic activities but only block-structured modeling constructs. It could be used as a structure template to avoid repeated modeling work. For example, a process modeler may have noticed that the structure as shown in Listing 4.10 has been frequently needed in modeling BPEL processes. Reusing it allows process modelers to fill the structure with BPEL activities that are needed for different use cases without having to draw the structure again.

```
<scope>
  <eventHandlers>
  </eventHandlers>
  <faultHandlers>
    <catchAll>
    </catchAll>
  </faultHandlers>
  <compensationHandler>
  </compensationHandler>
  <flow>
    <sequence>
    <\sequence>
    <sequence>
    <\sequence>
  </flow>
</scope>
```

Listing 4.10 A snippet of a BPEL process that does not contain at least one basic activity should not be considered as a valid BPEL fragment.

Chapter 5
Extracting BPEL Fragments

This chapter provides a mechanism for extracting BPEL fragments from existing BPEL processes. In Section 5.2 we introduce disjoint selection classes and extraction modes that will influence the results of extracting the selected process activities. In Section 5.3 we present the algorithms for extracting the selected process activities according to their respective extraction mode. Section 5.4 provides algorithms for reducing redundant process elements in the resulting BPEL fragment.

5.1 Introduction

In this chapter we present a mechanism for extracting arbitrary parts as a reusable BPEL fragment from an existing BPEL process. The extraction mechanism comprises three phases: the selection phase, the construction phase, and the reduction phase.

In the selection phase, process modelers select the process activities that should be included in the BPEL fragment. In addition, they also have the possibility to specify, in which mode the selected process activities should be extracted, i.e. whether the original control dependencies between the selected process activities should be retained in the resulting BPEL fragment or not.

In the construction phase, selected process activities are extracted in the respective mode that the process modeler chose in the selection phase. Depending on the extraction mode, opaque activities may be used to replace the not selected process activities between the selected ones, so that the original control dependencies of the selected ones can be retained.

In the reduction phase, redundant process elements that do not belong to the original selection of the process modeler are removed, such as opaque activities that are generated during the construction phase, empty structured activities, etc. The reduction preserves the original control dependencies of the remaining process activities in the resulting BPEL fragment.

5.2 Selection Phase

The selection phase is the starting point to extract a BPEL fragment. In this phase, a process modeler selects which BPEL constructs should be extracted and specifies how the selected elements should be included in the BPEL fragment being designed. The selections can be conducted either (semi)automatically or manually.

In a (semi)automatic manner, process modelers could extract BPEL fragments by using an SQL-like query. For example, if a process modeler wants to extract a BPEL fragment that contains all the activities that are dealing with invoicing, then he can execute the following query: *SELECT activities WHERE partnerLink.name="invoicing"*. For more sophisticated queries, process modelers may have to tag the process artifacts with additional metadata, which can then be used in the statements for extraction. The (semi)automatic manner is out of the scope of this thesis. An exemplary implementation can be found in [141]. The (semi)automatic manner requires that all the needed information for the query is available. If the BPEL constructs to be selected do not share common properties, then the query request could be complex and unreadable for a human being. Thus, the (semi)automatic manner is suitable for selecting activities that share common properties and the properties are already captured and made available.

In the manual manner, process modelers manually select the BPEL constructs that should be included in the target BPEL fragment. The manual manner can be used as a complementary mechanism for (semi)automatic selections.

Regardless how the selections have been made, process modelers have to specify in which extraction mode the selected process activities should be extracted (see Section 5.2.2).

5.2.1 Disjoint Selection Classes

To extract a BPEL fragment process modelers have to select which elements in the existing BPEL process should be included in the BPEL fragment. Different selections represent four disjoint classes for a given complex construct: envelope selection, complete selection, partial selection, and inner selection (Figure 5.1). Boldfaced lines in the figure demonstrate the selections.

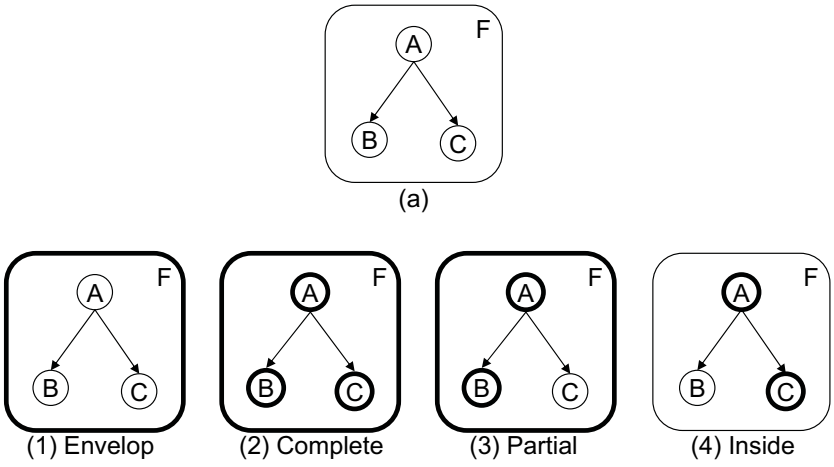


Fig. 5.1 Disjoint Selection Classes.

Envelope Selection

For a given complex construct, an *envelope selection* contains only the complex construct itself without selecting any of its enclosed activities. An envelope selection is very useful to design a BPEL fragment that represents structural patterns.

For example, a process modeler may have identified that the following BPEL fragment represents a frequent process pattern (Figure 5.2). The sample fragment gets the customer information and use it to route the control flow based on this information. For example, for order-to-cash processes it decides on the execution path based on the customer category, e.g. cooperate customer, premium customer, or regular customer; while for invoicing processes the execution path is chosen based on payment methods and the respective credit history. Besides the transition conditions the process logic of each branch is also different. As the process modeler needs the combination of the activity *getCustomerInfo* and the following `<if>` activity very often, it represents a frequent pattern. Therefore, the process modeler wants to extract it as a BPEL fragment to avoid repeated modeling work. To do so the process modeler can select the activity *getCustomerInfo* and the `<if>` activity without any of its constituents.

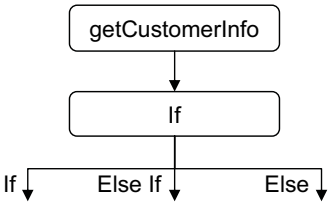


Fig. 5.2 A BPEL fragment with an empty *if* activity.

Complete Selection

For a given complex construct a *complete selection* contains both the complex construct itself and all its enclosed elements. To make a complete selection, a process modeler selects the complex construct and all the enclosed elements of the complex construct. However, this approach would be cumbersome; especially when the complex constructs contains a great number of elements.

An alternative approach would be to allow process modelers to select only the complex construct itself as a shortcut to make a complete selection. But only selecting the complex construct itself is ambiguous in this case: does the process modeler mean an envelope selection or a complete selection? To avoid this ambiguity the implementation should ask the process modeler for his/her intension before extracting the selected elements.

When using the shortcut to make a complete selection all the enclosed elements will inherit the extraction mode of the complex construct on which the shortcut selection has been made. If a process modeler needs to extract the complex construct in a different extraction mode than its enclosed elements, then the shortcut cannot be used. In this case, the process modeler has to select the enclosed elements and assign the required extraction mode to them. Also the extraction mode of the enclosing complex construct could be inherited by the enclosed elements until certain nesting level. Enclosed elements that are beyond the specified nesting level require explicit assignment of extraction mode.

Partial Selection

For a give complex construct a *partial selection* contains the complex construct itself and a proper subset of its immediately enclosed elements.

Inner Selection

For a given complex construct an *inner selection* does not contain the complex construct itself but a subset of its immediately enclosed elements. Depending on the extraction mode and the selections, an inner selection may result in

the same BPEL fragment as an partial selection. We will discuss this later in Section 5.2.2 Extraction Modes.

5.2.2 Extraction Modes

Extraction modes define how the control dependencies of the selected activities should be handled during the extraction. In this section we introduce two extraction modes: the connected mode and the isolated mode. The extraction algorithm will be discussed in Section 5.3.

Connected Mode

The connected mode enables process modelers to extract parts of an existing BPEL process while retaining the original control dependencies between the enclosed process activities in the BPEL fragment. BPEL integrates two different process modeling approaches: block-structured and graph-based modeling approaches.

In the block-structured process modeling approach, control dependencies are specified solely by the immediate enclosing construct. In this case, whether the original control dependencies of the immediate enclosed process activities can be retained after the extraction in the connected mode depends on the selection of the process modeler. If the process modeler selects the block-structured construct, then the control dependencies of its immediate enclosed process activities are retained. Otherwise, they are lost. The reason why we allow the latter case is that, we want to enable process modelers to make delayed decisions on the control dependencies of the selected process activities at the time of reuse.

In the graph-based modeling approach, control dependencies are explicitly specified by using BPEL links. When extracting process logic that are modeled using the graph-based approach links are extracted with the selected process activities. Intermediate modeling constructs that are not explicitly selected by the

process modeler may be needed in order to retain the original control dependencies.

For example, a process modeler wants to extract the activities X, A, and C in connected mode (Figure 5.3 (a)). Although the flow activity F itself has not been selected explicitly, it is included in the BPEL fragment in order to retain the control dependencies between the activity X and A (see Figure 5.3 (b)).

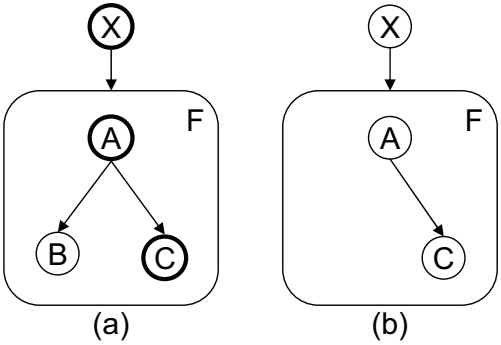


Fig. 5.3 The flow activity F is needed to retain the original control dependencies of the selected activities.

An alternative is to remove the flow activity F and to connect the activity X with the activity A. However, this would change the original process structure of selected activities. One of our design goals is to retain the original process structure during the extraction phase as much as possible, so that process modelers can easily recognize the BPEL fragments that they have extracted. For that reason we do not consider this alternative in this thesis.

Isolated Mode

In the isolated mode the extraction mechanism eliminates completely the control dependencies between the selected activities. Each selected activity that should be extracted in the isolated mode is placed as an immediate enclosed activity of the root element in the BPEL fragment being designed.

For example, a process modeler may want to extract the activities *A* and *B* from the BPEL snippet shown in Figure 5.4 for reuse. The activities can either be reused in a `<flow>` activity or a `<sequence>` activity. The control dependencies between *A* and *B* should be determined at the time of reuse. If we extract activities *A* and *B* in the connected mode without having selected the `<flow>` activity, then we get a BPEL fragment, in which the control dependencies *A* and *B* are retained. The resulting BPEL fragment does not meet our needs. To enable this scenario we introduce the isolated mode. Extracting *A* and *B* in the isolated mode results in a BPEL fragment as shown in Figure 5.4.

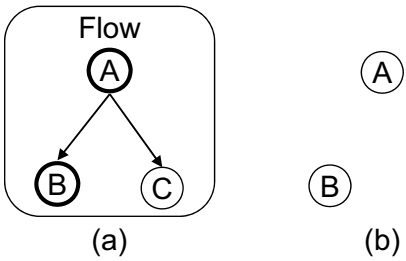


Fig. 5.4 Extracting activities *A* and *B* in isolated mode.

5.2.3 Retaining Process Structure

In the present modeling approach, a process modeler selects only the BPEL constructs that should be included in the BPEL fragment being designed. However, in certain use cases, this approach retains the required process structure during the extraction.

Let's consider an example: A process modeler wants to extract a part of a BPEL process as a BPEL fragment. The extraction should preserve the same process structure as shown in Figure 5.5, which represents a frequent pattern for modeling four-eyes-principle. In addition, the BPEL fragment should contain the activities *check4EyesPrinciple* and *escalate*, because they are mandatory constituents for modeling four-eyes-principle in the organization. The activities

costClaim, *internalRevision*, and *certificatAssignment* are optional and can be substitued by other activities. Therefore, they should not be included in the target BPEL fragment. In other words, the process fragment should contain the activities *check4EyesPrinciple*, *escalate*, and the links l_1, l_2, l_3, and l_4.

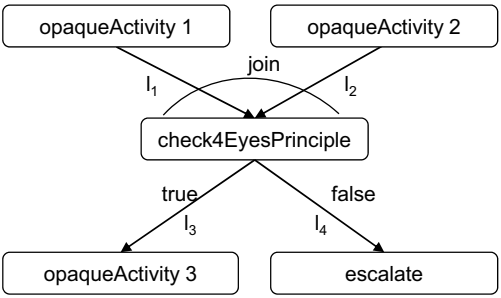


Fig. 5.5 A part of a BPEL process that models the four eyes principle.

In order to retain the original control flow, the fragment should be extracted in the connected mode. As we discussed before, to extract a BPEL fragment process modelers have to select the constructs that should be included in the resulting BPEL fragment. If the process modeler selects the activities *check4Eyes-Principle* and *escalate*, then the resulting fragment (Figure 5.6) would only contain the selected activities. The reason is that unselected activities will be replaced by opaque activities in the construction phase, which in turn will be removed in the reduction phase. Alternatively, if the process modeler selects all five activities, then the resulting fragment would also contain activities *costClaim*, *internalRevision*, and *certificatAssignment* which are not required. Thus, we need an approach for retaining process structures of BPEL fragments without having to include any unneeded BPEL constructs.

5.2.3.1 Using Opaque Activity to Retain Process Structure

Opaque activities are used as explicit placeholders in BPEL. It represents an ideal construct to hide unneeded activities in a BPEL fragment.

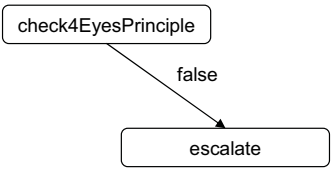


Fig. 5.6 Only extracting the required process activities in the connected mode cannot retain the original process structure.

In the example above, the process modeler can replace the activity *costClaim*, *internalRevision*, and *certificateAssignment* with opaque activities and then extract them together with the activity *check4EyesPrinciple* and *escalate* in the connected mode. The resulting fragment is shown in Figure 5.7.

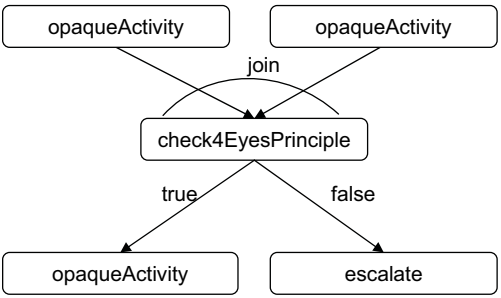


Fig. 5.7 Using opaque activities to retain the original process structure.

We call the opaque activities that are explicitly modeled by process modelers *modeled opaque activities*. To the contrary, opaque activities that are generated during the construction phase to retain the original control dependencies are called *generated opaque activities*. Generated opaque activities can be removed in the reduction phase, while modeled opaque activities cannot.

5.2.3.2 Using Incomplete Link to Retain Process Structure

Although using opaque activities in the selection phase can retain the required process structure during the extraction, opaque activities per definition have

restricted semantics: an opaque activity represents an explicit placeholder for exactly one BPEL activity. But, under certain circumstances process modelers may want to replace an opaque activity with a BPEL fragment.

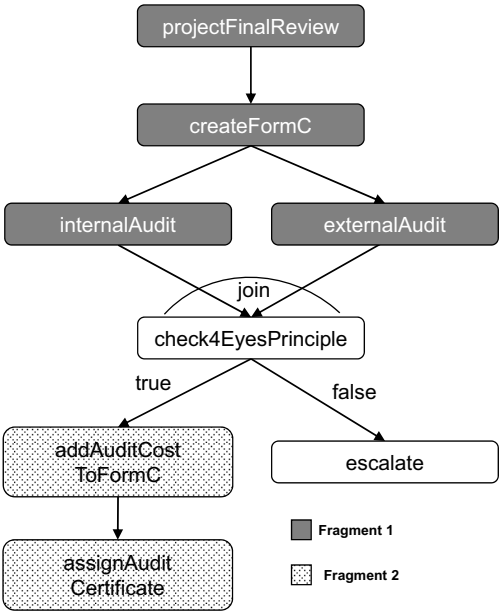


Fig. 5.8 An exemplary use case shows that a process modeler may want to replace the placeholder with more than one BPEL activities.

For example, a process modeler wants to reuse the process fragment shown in Figure 5.8. He may want to replace the opaque activities with reusable BPEL fragments. However, this is not allowed due to the strict semantics of opaque activity. In this case, we can use incomplete links (see Section 4.5.2.2) to retain the required process structure.

To do so, the process modeler first selects the activities in Figure 5.8 that should be included in the BPEL fragment, i.e. *check4EyesPrinciple* and *escalate*. After that the process modeler selects the two incoming links of the activity *check4EyesPrinciple* and the outgoing link of the activity *check4EyesPrinciple*.

As the source activities of the incoming links of the activity *check4Eyes-Principle* are not selected, they will be replaced by opaque activities in the construction phase. As the opaque activities are generated opaque activities, they will be removed in the reduction phase. Their incoming and outgoing links will also be removed, if they have any. Therefore, the explicit selection of the links is necessary to tell the extraction algorithm to keep them in the reduction phase. The resulting BPEL fragment is shown in Figure 5.9.

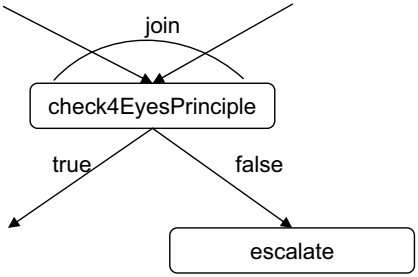


Fig. 5.9 Using incomplete links to retain the original process structure.

5.2.3.3 Using Bag Activity to Retain Process Structure

An alternative to the usage of incomplete links for retaining process structure is to use the `<bagActivity>`. Both constructs can be used to model unknown process logic in a BPEL fragment. The difference between using a bag activity and an incomplete link is that a bag activity can be used to model unknown process logic while avoiding splitting a process fragment into several parts. Let's consider the following example in Figure 5.10.

In the figure a) the process modeler uses a bag activity to model unknown process logic. It is clear that some unknown process logic should take place after the activity *getCustomerInfo* and that the activity *recordResult* should be performed immediately after the unknown process logic, if the transition condition evaluates to be `true`. When the original control dependencies of the

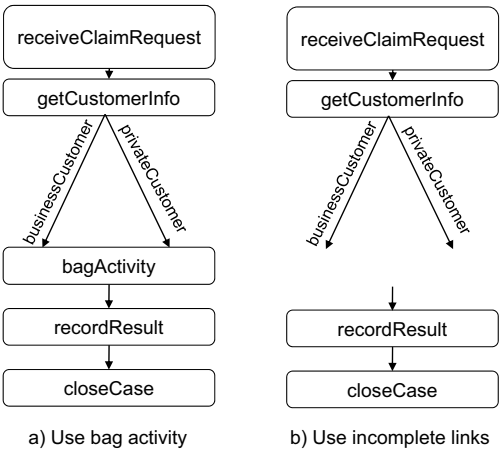


Fig. 5.10 A bag activity acts as an intermediate part of a process fragment.

remaining activities should be retained, process modelers should follow this approach.

In the figure b), the process modeler uses incomplete links to model unknown process logic. The control dependencies between the two isolated parts are not specified. It is not clear which part should be performed before the other one. But it allows process modelers to decide on the control dependencies of the parts at the time of reuse. Thus, if delayed decision on control dependencies of the parts in a process fragment is required, process modelers should choose this approach.

5.3 Construction Phase

The construction phase conducts the extraction of the BPEL fragment based on the selections and the extraction modes that the process modeler specified in the selection phase. We use the BPEL process illustrated in Figure 5.11 for our discussion in this section. The illustration uses different borders to distinguish the different extraction modes of the selections as well as the unselected elements in the BPEL process. Simple border lines denotes the unselected process ele-

ments; boldfaced lines highlight the selected elements that should be extracted in the connected mode; double border lines show the selected elements that should be extracted in the isolated mode; dashed border lines represents the selected elements that should be extracted in the isolated mode. The labeling in braces shown in the BPEL process are the abbreviations that we will use later in this section.

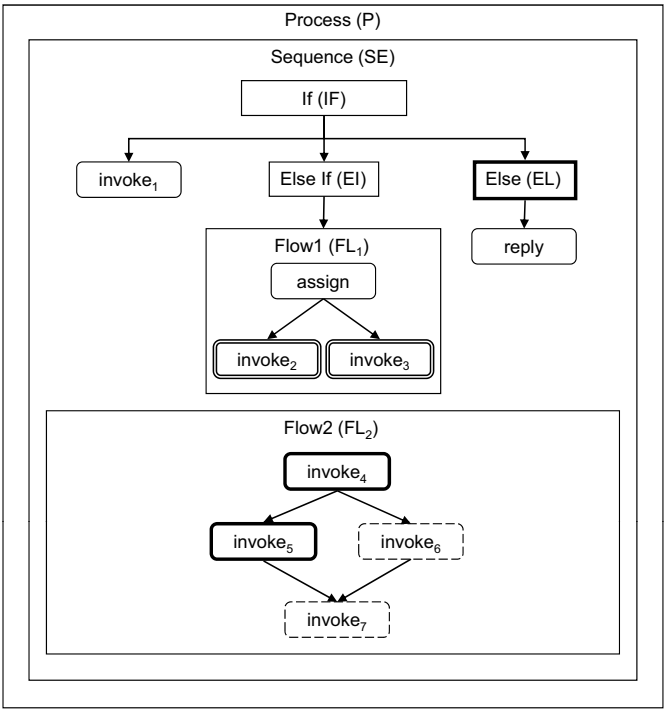


Fig. 5.11 An example of a BPEL process. Simple border lines denotes the unselected process elements; boldfaced lines highlights the selected elements that should be extracted in the connected mode; doubled border lines shows the selected elements that should be extracted in the isolated mode; dashed border lines represents the selected elements that should be extracted in the stand-alone mode. The labeling in the braces shown in the BPEL process are the abbreviations that we will use later in this section.

5.3.1 Extract in Connected Mode

In the connected mode, the resulting BPEL fragment must contain all the selected BPEL constructs for the extraction in connected mode while retaining their original control dependencies. An intuitive solution would be to find the minimal connected subgraph of the process graph that contains all the selected BPEL constructs. However, finding the minimal subgraph that contains a given set of nodes in a DAG is equivalent to the Steiner tree problem [72], which is NP-complete.

In BPEL, each element is located in a nesting construct. The utmost nesting construct is the `<process>` element. If we can find a nesting construct that contains all the selected elements, then we can extract the nesting construct with all its enclosed elements. The not selected activities will then be replaced by opaque activities, which can be removed in the reduction phase, if needed. To realize this approach we have to find the lowest common nesting construct which contains all the selected elements.

Definition 4. Lowest Common Nesting Construct (LCNC)
For a given set of BPEL constructs S, a complex construct x is the lowest common nesting construct of S, if x satisfies the following conditions:

- x encloses all the constructs in S;
- let R be the set of all complex constructs that enclose all the elements in S, then $\forall y \in R$, y encloses x. □

The lowest common nesting construct can be efficiently computed with the help of the Nesting Hierarchy Tree (NHT).

5.3.1.1 Nesting Hierarchy Tree (NHT)

The nesting hierarchy tree of a BPEL process is built based on the nesting relations between the complex constructs used in the BPEL process. A nesting relation is defined as follows:

Definition 5. Nesting Relation
A nesting relation is a binary relation $\Gamma \subseteq C \times C$, where C denotes a set of

complex constructs. We say a complex construct c_1 is in a nesting relation with a complex construct c_2, only if c_1 encloses c_2. The nesting relation Γ is an irreflexive, asymmetric, and transitive relation:

- irreflexive: $\forall c \in C$ it holds $(c,c) \notin \Gamma$;
- asymmetric: $\forall x$ and $y \in C$, if $(x,y) \in \Gamma$, then $(y,x) \notin \Gamma$;
- transitive: $\forall x$, y, and $z \in C$, if $(x,y) \in \Gamma$ and $(y,z) \in \Gamma$, then $(x,z) \in \Gamma$. □

In a nesting hierarchy tree, nodes represent instances of complex constructs that are used in a given BPEL process, and edges represent the nesting relations between the complex constructs.

Definition 6. Nesting Hierarchy Tree (NHT)
A nesting hierarchy tree of a BPEL process is a directed tree $T = (V,E)$, where V is the set of instances of complex constructs used in the BPEL process, and E is the set of directed edges represent the nesting relations between the complex constructs. A directed edge $(u,v) \in E$ only if the complex construct u immediately encloses the complex construct v. □

The level of a node v in a NHT is the length of the longest path from the root r to v. We use the notion $level(v)$ to denote the level of the node v in the NHT. The level of the root r is zero. For a node v with $level(v) = i$ with $i \in \mathbb{N}$ we also say that the node v is at level i. The leaf nodes in a NHT represent complex constructs in a BPEL process that contain no further complex constructs but only basic activities. Algorithm 1 presents the algorithm for creating the nesting hierarchy tree with a depth-first approach.

The Algorithm 1 works in a recursive manner. It begins with a complex construct c. At the beginning of the algorithm, the node set V of the NHT is empty. If the construct c is a complex construct, then c is added to V. Then the algorithm recursively examines each immediately enclosed element x of c. If x is a complex construct, then x is added to V. As c immediately encloses x, an edge (c,x) that represents the nesting relation is created and added to the set E.

Let $P = (V_P, E_P)$ be the process graph. The worst-case time complexity of creating the Nesting Hierarchy Tree is $O(max(\{|V_P|, |E_P|\}))$. Note that as the process graph P is a weakly connected DAG, it has at least $|V_P| - 1$ edges. Thus, the worst-case run time of the algorithm is $O(max(\{|V_P|, |E_P|\})) \approx O(|E_P|)$.

Algorithm 1: creatNHT(c)

```
1 begin
2     if c is a complex construct then
3         add c to V;
4     forall the immediately enclosed constructs x of c do
5         if x is a complex construct then
6             add x to V;
7             add (c,x) to E;
8             createNHT(x);
```

When applying the Algorithm 1 on the BPEL process in Figure 5.11 we get the nesting hierarchy tree as shown in Figure 5.12.

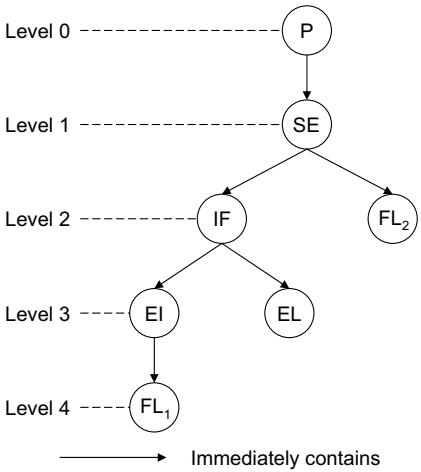


Fig. 5.12 The nesting hierarchy tree of the BPEL process *P* shown in Figure 5.11. The directed edges in this figure represent the nesting relations between the complex constructs.

5.3.1.2 Finding The Lowest Common Nesting Construct

A lowest common nesting construct is defined as follows:

Definition 7. Lowest Common Nesting Construct (LCNC)
For a given set of BPEL constructs $S \subseteq V$ of a NHT $T = (V, E)$, a complex construct $x \in V$ is a lowest common nesting construct, if x satisfies the following conditions:

- x is a common nesting construct of S: x contains all the BPEL constructs in S, i.e. $\forall s \in S$ there exists a path $(x, ..., s) \in T$;
- let R is the set of other common nesting constructs of S, then $\forall y \in R$ we have $level(y) < level(x)$. □

Before we present the algorithm for finding the lowest common nesting construct, we introduce the lemma and theorems that are used as the basis of our algorithm.

As BPEL is a XML representation, the XML elements used in the BPEL language must be well-formed. Therefore, we can derive Lemma 1.

Lemma 1. *For a given BPEL construct c in a BPEL process, there exists exactly one complex construct that immediately encloses c. We call the enclosing complex construct the parent nesting construct of c.* □

Theorem 1. *Each node that is not the root of a NHT has exactly one incoming edge.*

Proof. Assume to the contrary that there exists a node w in a NHT which has two incoming edges (u, w) and (v, w). According to the definition of NHT, an edge (u, w) denotes that u immediately encloses w. Analogously, (v, w) denotes that v immediately encloses w. Then there exist two complex constructs u and v that immediately enclose w, which contradicts Lemma 1. □

Theorem 2. *For a given BPEL construct c in a BPEL process, let x and y be complex constructs. If both* $(x, c) \in \Gamma$ *and* $(y, c) \in \Gamma$*, then either* $(x, y) \in \Gamma$ *or* $(y, x) \in \Gamma$.

Proof. Assume to the contrary that if $(x, c) \in \Gamma$ and $(y, c) \in \Gamma$ then neither $(x, y) \in \Gamma$ nor $(y, x) \in \Gamma$. As x encloses c, there exists a path p_1 from x to c with $(x, v_i, ..., v_{i+k}, c)$ and y is not in p_1 . Also, there exists a path p_2 from y to c with $(y, v_j, ..., v_{j+l}, c)$ and x is not in p_2. As both $(x, c) \in \Gamma$ and $(y, c) \in \Gamma$, the two

paths must have an intersection point, i.e. there exists either a node v_m in p_1 and a node v_n in p_2 with $v_m = v_n$ or the node c is the intersection point. In the first case, the node v_m has two incoming edges, i.e. (v_{m-1}, v_m) and (v_{n-1}, v_m). In the second case, the node c has two incoming edges, i.e. (v_{i+k}, c) and (v_{j+l}, c). Both cases contradict Theorem 1. □

Theorem 3. *For a given set of complex constructs S, there exist exactly one lowest common nesting construct.*

Proof. Assume to the contrary that there exist two lowest common nesting constructs x and y for a given set of complex constructs S. Then both x and y contain all the elements in S. Especially, there exists an $s \in S$ with $(x,s) \in \Gamma$ and $(y,s) \in \Gamma$. According to Theorem 2, it applies either $(x,y) \in \Gamma$ or $(y,x) \in \Gamma$. If $(x,y) \in \Gamma$, then y is not a lowest common nesting construct; if $(y,x) \in \Gamma$, then y is not a lowest common nesting construct. Both cases contradict our assumption. □

Theorem 4. *Two complex constructs x and y that are at the same level in the nesting hierarchy tree do not have a nesting relation. In other words, if* $level(x) = level(y)$*, then neither* $(x,y) \in \Gamma$ *nor* $(y,x) \in \Gamma$.

Proof. Assume to the contrary that $level(x) = level(y)$ and $(x,y) \in \Gamma$. According to the definition of a NHT, an edge in a NHT represents the containing relation of two complex constructs, there must exist a path p from x to y with the length $length(p) = level(y) - level(x)$, where $length(p) \geq 1$. As y has exactly one incoming edge, the path over x is the only path from the root to y. Thus, the longest path from the root to y must be $level(x) + length(p)$, which contradicts our assumption of $level(x) = level(y)$. Analogously, y does not enclose x. □

Theorem 5. *For a given set of constructs* $S \subseteq V$ *of a NHT* $T = (V,E)$*, let c be the lowest common nesting construct of S. Then* $\forall s \in S$ *it applies* $level(s) \geq level(c)$.

Proof. Assume to the contrary that c is the lowest common nesting construct of S and there exists an $s \in S$ with $level(c) > level(s)$. As $level(c) > level(s)$, the complex construct c does not enclose the complex construct s. Otherwise,

there must exist a directed edge $(c,s) \in E$ and $level(c) < level(s)$. Therefore, c is not the lowest common nesting construct of all the constructs in S, which is in contradiction to our assumption. □

The Algorithm

The algorithm of finding the lowest common nesting construct for a set of BPEL constructs in a BPEL process is developed based on the theorems we introduced above. Here we present a summary of how the algorithm works. It takes a set of selected constructs S that should be extracted in the connected mode as input. As S may contain basic activities at the beginning, we invoke the algorithm with $lowestLevel = -1$ so that the algorithm knows it should examine whether S contains basic activities. In case S contains basic activities, the algorithm begins with replacing each basic activity with its parent nesting construct x (Algorithm 2, line 4-9).

After having replaced the basic activities the algorithm tries to find the lowest common nesting construct. Algorithm 2 is a recursive algorithm, which repeats until S has exactly one element.

In case S contains no basic activities, we compare the level of each complex construct and keep the lowest level of the complex constructs in the variable $lowestLevel$.

If S contains exactly one complex construct, then it applies Theorem 3. Thus, this complex construct is the lowest common nesting construct of all the originally selected elements (Algorithm 2, line 14-15).

If S contains more than one element, then the lowest common nesting construct could be either in S or an ancestor of all the elements in S. In fact, the lowest common nesting construct locates at least at the lowest level of all the elements in S (Theorem 5). The complex constructs in S may be located at different levels in the nesting hierarchy tree. Thus, for each element in S whose level is greater than the lowest level ($lowestLevel$) of the elements in S we replace it with its ancestor nesting construct, whose level is at the lowest level $lowestLevel$. As S is a set, S has no duplicate elements. Therefore, if some elements in S have the same ancestor at level $lowestLevel$, the ancestor appears

Algorithm 2: findLCNC(S, *lowestLevel*)

```
1  S: a set of constructs;
2  lowestLevel: the lowest level of the elements in S;
3  begin
4      if lowestLevel = −1 then
5          forall the s in S do
6              if s is a basic activity then
7                  x = the parent nesting construct of s;
8                  remove s from S;
9                  S = S ∪ {x};
10             if level(x) < lowestLevel then
11                 lowestLevel = level(x);
12         findLCNC(S, lowestLevel);
13     else
14         if |S| = 1 then
15             return s in S ;   /* s is then the lowest common
                   nesting construct */
16         else
17             forall the s in S do
18                 while level(s) > lowestLevel do
19                     x = the parent nesting construct of s;
20                     remove s from S;
21                     S = S ∪ {x};
22                 lowestLevel = lowestLevel − 1;
23             findLCNC(S, lowestLevel);
24 end
```

only once in S. Now all the elements in S are at the same level and the algorithm begins with a new recursion by invoking itself ((Algorithm 2, line 23).

If S still contains more than one element, then none of them is the lowest common nesting construct. As for any two complex constructs at the same level of the nesting hierarchy tree they do not have a nesting relation (Theorem 4). Thus, the lowest common nesting construct of them must locate at the lower level in the nesting hierarchy tree. Therefore, we decrease the lowest level by 1 and recursively invoke the Algorithm 2 again.

If S contains exactly one element s, then s is the lowest common nesting construct of all the originally selected constructs (Theorem 3). As s is the ancestor of all the originally selected constructs except itself, it encloses directly or indirectly all of them. Therefore, s is a common nesting construct of the other complex constructs. And for any complex construct x that encloses all the elements in S, it must enclose s. Thus, x must be higher in the hierarchy than s, i.e. $level(x) < level(s)$. Therefore, s is the lowest common nesting construct.

Let's consider the example shown in Figure 5.11 again. As highlighted in Figure 5.11 the process modeler has selected to extract the whole `<else>` branch and the activities *invoke4* and *invoke5* in the connected mode. Thus, the initial set S contains the three elements $\{Else, invoke4, invoke5\}$.

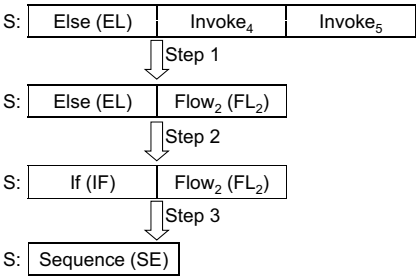


Fig. 5.13 An example of finding the lowest common nesting construct using the Algorithm 2.

Step 1: we replace the basic activities in S with their parent nesting constructs. Both *invoke4* and *invoke5* have the same parent nesting construct $Flow_2$. The resulting set S is then $\{Else, Flow_2\}$.

Step 2: we replace the elements in S with their ancestors at the level $lowestLevel$. In the nesting hierarchy tree of the process P in Figure 5.12, the element $Else$ is located at level 3 while the element $Flow_2$ is located at level 2. The lowest level of them is then level 2. Therefore, we replace the element $Else$ with its ancestor at level 2 in the nesting hierarchy tree. The ancestor of $Else$ at level 2 is the If activity. After the replacement $S = \{If, Flow_2\}$. All of the elements in S are now at the same level in the nesting hierarchy tree. But S contains more than one elements, thus, we have to continue with our algorithm.

Step 3: we continue our search to the next higher level, i.e. level 1. We replace all the elements in S with their parent nesting construct. Both If and $Flow_2$ have the same parent nesting construct, which is *Sequence* at level 1. After the replacement S contains exactly one complex construct. And the complex construct *Sequence* is the lowest common nesting construct of the selected elements $\{Else, invoke_4, invoke_5\}$.

After having found the LCNC, i.e. the <if> activity, it will be extracted with all its constituents from the original process and placed as an immediate child element within the <fragment> element.

5.3.2 Extraction in Isolated Mode

In the isolated mode, each selected element will be placed as an immediate child element within the root element of the BPEL fragment apart from some exceptions. BPEL uses activity containers to logically group activities together. Some of these activity containers are not allowed used as immediate child elements of the <fragment> element. They include:

- <else> and <elseIf> of an <if> activity;
- <onMessage> and <onAlarm> of a <pick> activity;
- <catch> and <catchAll> of a <faultHandlers>;
- <onEvent> and <onAlarm> of an <eventHandlers>.

Thus, if a process modeler wants to extract these activity containers in the isolated mode, the extraction algorithm will include both the immediate enclosing construct and the activity container in the target fragment.

However, <faultHandlers> and <eventHandlers> can be used once as an immediate child element of the <fragment> element. As discussed before, when extracting their activity containers in the isolated mode, the algorithm will also include the <faultHandlers> or <event− Handlers> in the target BPEL fragment. If the BPEL fragment already contains the <faultHandlers> element, then the selected <catch> and <catchAll> containers will be placed in the existing <faultHandlers> element. In

this case we can avoid the duplication of the <faultHandlers> element. When extracting selected <catch> and <catchAll> elements, the extraction algorithm may examine whether there are duplicated fault handlers in the <faultHandlers> element, e.g. based on the qualified fault name. This may require domain-specific knowledge and is out of the scope of this thesis and its prototypical implementation. If the BPEL fragment contains no <faultHandlers> element, then the extraction algorithm will add a <faultHandlers> element within the root element <fragment> and place the selected <catch> and <catchAll> in it. The same also applies for event handlers.

Compensation and termination handlers can also be used once as immediate child element of the <fragment> element. If a process modeler wants to extract a compensation handler or a termination handler in the isolated mode, the extraction algorithm should also examine possible duplication. As a compensation handler allows exactly one activity as its immediate child element, it is not possible to add the immediate child activity to the existing compensation handler. Therefore, if the BPEL fragment contains already a compensation handler, then the modeling tool should ask the process modeler whether the existing compensation should be replaced by the new one. If the BPEL fragment contains no compensation handler as its immediate child element, then extraction algorithm places the selected compensation handler as an immediate child element of the <fragment> element. The same also applies for termination handler.

5.4 Reduction Phase

The constructed BPEL fragment may contain generated opaque activities. Process modelers may want to (i) retain or (ii) remove the generated opaque activities in the BPEL fragment.

In case (i), process modelers may want to keep the generated opaque activities in the resulting BPEL fragment. In this case, the BPEL fragment can be reused as a template. The generated opaque activities hide certain process de-

tails that allow completing and adding execution behavior at the time of reuse. For this use case, process modelers may also define a profile for the usage of the BPEL fragment as a template[1].

In case (ii), generated opaque activities are considered as redundant. They make a BPEL fragment verbose and hard to read. In this case, the process modeler may want to remove the redundant opaque activities. Cleaning the redundant generated opaque activities makes the BPEL fragment more compact, therefore, enables process modelers to concentrate on the essential process logic.

The reduction phase removes the redundant generated opaque activities in a BPEL fragment (case (ii)). Because we want to enable process modelers also to use the resulting BPEL fragment without removing the generated opaque activities (case (i)), the reduction phase has been considered as an optional phase.

5.4.1 Reduction Rules

We introduce the reduction rules as the basis of the reduction algorithms for removing generated opaque activities. Considering the incoming and outgoing links, generated opaque activities can be classified into three categories: isolated opaque activities, terminal opaque activities, and inner opaque activities.

Isolated opaque activities have neither incoming nor outgoing links. Terminal opaque activities include entry opaque activities and exit opaque activities. Entry opaque activities have only outgoing links, while exit opaque activities have only incoming links. Inner opaque activities have both incoming and outgoing links. We call terminal and inner opaque activities non-isolated opaque activities.

The reduction rules do not consider the operational semantics of the activities (e.g. transition conditions, join conditions, and dead path elimination), but focuses solely on the syntactical process logic. When removing a link, its tran-

[1] similar motivation for the abstract process profile for template specified in the BPEL standard

sition condition will also be removed. When removing a generated opaque activity with a join condition, the join condition will also be removed.

We use *o* to indicate modeled opaque activity, *b* to indicate bag activity, *g* to indicate generated opaque activity, *p* to indicate the preceding activity of *g*, and *s* to indicate the succeeding activity of *g*.

5.4.1.1 Isolated Opaque Reduction Rule

The isolated opaque reduction rule is introduced to remove generated opaque activities that have no incoming and outgoing links. Let's consider the example shown in Figure 5.14. A process modeler selects the sequence activity and two of its enclosed activities *A* and *B* and wants to extract them in the connected mode (denoted by the bold border lines in Figure a)). The lowest common ancestor of all the selected activities is the sequence activity itself. In this case, the algorithm extracts the complete sequence activity including all its enclosed activities. The not selected activity *C* is then replaced by a generated opaque activity, which is an isolated generated opaque activity (Figure b)).

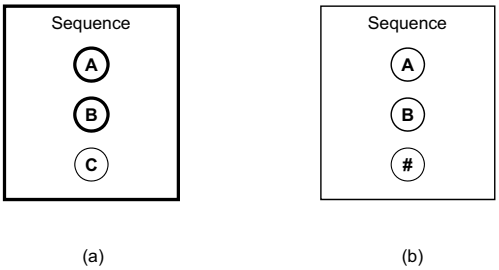


Fig. 5.14 As the not selected activity *C* has neither incoming nor outgoing links, the generated opaque activity that replaces it is an isolated generated opaque activity.

In a graph-structured construct the control dependencies are explicitly specified by links. Removing the generated opaque activity that has no incoming and outgoing links does not change the original control dependencies of the remaining activities.

In a block-structured construct the control dependencies are defined by the order of the enclosed activities. As the generated opaque activity has no incoming and outgoing links, its control dependency is solely determined by its position in the block-structured construct. Removing it does not change the relation of its preceding and succeeding activities within the structured construct.

Therefore, we introduce the following rule:

Rule 1. If a generated opaque activity g has neither incoming nor outgoing links, then remove g. □

5.4.1.2 Non-Isolated Opaque Reduction Rules

As defined before a non-isolated opaque activity has either incoming links, or outgoing links, or even both. The basic idea to remove a generated non-isolated opaque activity g is to remove its incoming and outgoing links firstly, unless they have been selected in the selection phase (see Section 5.2.3.2 Retain Process Structure by Links). After that g becomes an isolated opaque activity. Then we can apply Rule 1 to remove it.

Before we begin with the discussion on reduction rules for removing incoming and outgoing links, we introduce some terms that we will use for the discussion.

We say an activity p is a preceding activity of an activity a, if a can only be started after p is completed. In other words, if there exists an execution path from p to a and p is not an enclosing construct of a, then p is a preceding activity of a. And a is a succeeding activity of p. We do not consider an enclosing activity of a as its preceding activity, as the start of a does not require the completion of the enclosing activity.

Let P be the set of all preceding activities of a, $p \in P$ is an immediate preceding activity of a, only if for all $p' \in P$ it applies $level(p) \leq level(p')$. Similarly, let S be the set of all succeeding activities of a, s is an immediate succeeding activity of a, only if for all $s' \in S$ it applies $level(s) \leq level(s')$.

When removing an outgoing link l of g we have to make sure that removing l does not change the control dependencies between the succeeding activity s (the target activity of l) and the remaining activities (except g). In fact, we have

to examine the following cases: (i) the control dependencies between s and its enclosed constructs; (ii) the control dependencies s and its enclosing constructs; (iii) the control dependencies between s and g's preceding activities; (iv) the control dependencies between s and its succeeding activities.

Case (i): removing l does not change the enclosing relationships between s and its enclosed constructs, if s is not a basic activity. If s is a basic activity, then s has no enclosed activities.

Case (ii): similarly, removing l also does not change the enclosed relationship between s and its enclosing constructs.

Case (iii): removing l could make s lose its control dependencies on g's preceding activities. The control dependencies are lost only if g has preceding activities within the Lowest Common Nesting Construct (LCNC) of g and s. Let's consider the example illustrated in Figure 5.15.

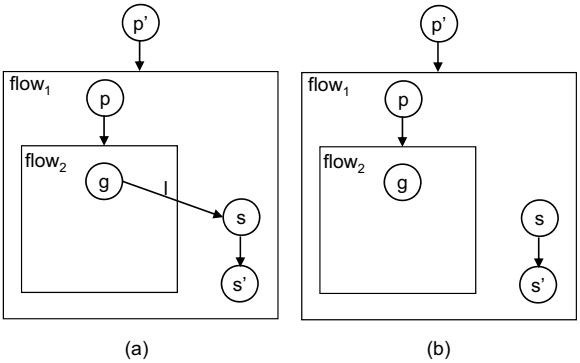


Fig. 5.15 Removing l will change the control dependencies between s and p, which is a preceding activity that is located within the LCNC of g and s.

In Figure 5.15 $flow_1$ is the LCNC of g and s. We can find two preceding activities of g in the example: p and p'. The activity p is the preceding activity of g that is enclosed within the LCNC of g and s, i.e. in $flow_1$. While p' is a preceding activity that is located outside the LCNC of g and s. When removing the outgoing link l of g, the activity s loses its control dependency on p. In the original process model p completes before s can start. After removing the link l, p becomes a parallel activity to s. Thus, removing l will change the control

dependencies between s and g's preceding activities that are located within the LCNC of g and s.

However, without the link l the original control dependencies of p' and s are retained. As removing the link l does not change the enclosing relationships, s is still enclosed within $flow_1$. The activity s can only starts, after $flow_1$ has been started. And $flow_1$ can only starts, after p' is completed. That means, regardless whether the link l has been removed, as long as s is enclosed within $flow_1$, its control dependency on p' does not change. Thus, removing l does not change the control dependencies between s and g's preceding activities that are located outside the LCNC of g and s.

Case (iv): removing l does not change the control dependencies between s and its succeeding activities. From the example shown in Figure 5.15 we can see that removing the link l does not change the original control dependencies between s and its succeeding activity s'.

According to the discussion above, removing the outgoing link l of g will makes s lose the control dependencies on g's preceding activities, only if g's preceding activities are located within the LCNC of g and s. Otherwise, removing l does not change the original control dependencies of the remaining activities.

The same discussion can be conducted for removing an incoming link of g. Now let l be an incoming link and p be a preceding activity of g (the source activity of l). Analogously, removing l will makes p lose the control dependencies on g's succeeding activities, only if g's succeeding activities are located within the LCNC of g and p. Otherwise, removing l does not change the original control dependencies of the remaining activities.

Based on the discussions we introduce in the following the reduction rules for removing incoming and outgoing links of a generated opaque activity.

Removing Outgoing Links

For a generated opaque activity g, let l be an outgoing link of g and s be the target activity of l.

According to the discussion above, removing l will change the control dependencies between s and g's preceding activities that are located within the LCNC of g and s. Thus, we consider the following two cases: (i) g has preceding activities within the LCNC of g and s; (ii) g has no preceding activities within the LCNC of g and s.

Rule 2. If a generated opaque activity g has no preceding activities within the LCNC of g and s, then remove l. □

As shown in Figure 5.16, g has no preceding activities within the LCNC of g and s, after removing the outgoing link l the original control dependencies of the remaining activities are retained.

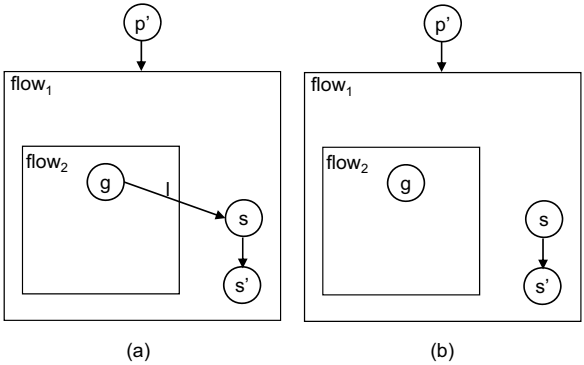


Fig. 5.16 If g does not have preceding activities within the LCNC of g and s, then removing l does not change the original control dependencies between s and the remaining activities except g.

Rule 3. If a generated opaque activity g has a preceding activity p within the LCNC of g and s and there exits an alternative path from p to s that does not contain g, then remove l. □

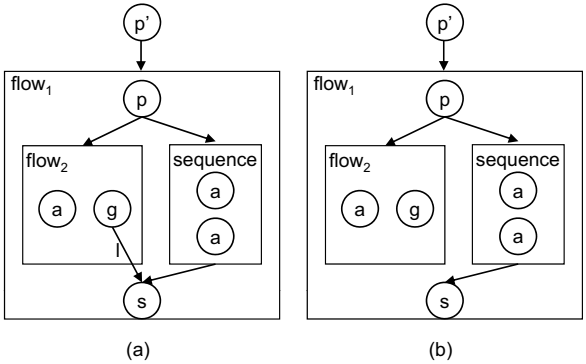


Fig. 5.17 Through the alternative path over the *sequence* activity, the original control dependency between p and s is retained.

As g has preceding activities within the LCNC of g and s, removing l will make s lost the original control dependency of p. However, there exists an alternative path from p to s that does not contain g, which means that the alternative path does not contain l. The original control dependencies between p and s is retained through the alternative path. In this case, the outgoing link can be removed.

Rule 4. If a generated opaque activity g has a preceding activity p within the LCNC of g and s and there exists no alternative path from p to s that does not contain g, then remove the link l and add a new link (p,s). □

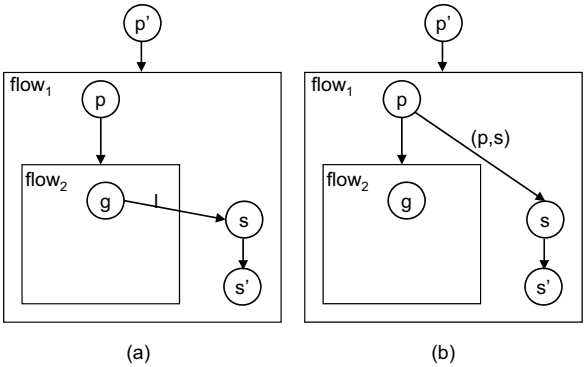


Fig. 5.18 As there exists no alternative path from p to s, we create a new link (p,s) after removing l to restore the original control dependency between p and s.

As there exists no alternative execution path from p to s that does not contain g, the control dependency between p and s depends solely on the link l. Thus, after removing the link l we have to rebuild the original control dependency between p and s. In the original process model, the activity s can only be started after the completion of p. To restore this control dependency without considering the original transition condition and data flow, we create a new link (p,s) after having removed l. The transition condition of the new link is set to $undefined$, as neither the original transition condition of p's outgoing link nor that of s's incoming link may apply. Thus, we let the process modeler redefine the transition condition.

Removing Incoming Links

For a generated opaque activity g, let l be an incoming link of g and p be the source activity of l.

Similarly, removing l will change the control dependencies between p and g's succeeding activities that are located within the LCNC of g and p. Thus, we consider the following two cases: (i) g has succeeding activities within the LCNC of g and p; (ii) g has no succeeding activities within the LCNC of g and

p. Rule 5 is introduced for case (i), while Rule 6 and Rule 7 are introduced for case (ii).

Rule 5. If a generated opaque activity g has no succeeding activities within the LCNC of g and p, then remove l. □

Rule 6. If a generated opaque activity g has a succeeding activity s within the LCNC of g and p and there exits an alternative path from p to s that does not contain g, then remove l. □

Rule 7. If a generated opaque activity g has a succeeding activity s within the LCNC of g and p and there exists no alternative path from p to s that does not contain g, then remove the link l and add a new link (p,s). □

5.4.2 Reduction Algorithms

The reduction algorithms are developed based on the reduction rules we introduced in the previous section. In this section we first introduce several supporting algorithms and then present the algorithms for removing generated opaque activities in modeling constructs of BPEL fragments.

5.4.2.1 Find Immediate Preceding Activities

As discussed before, to remove an outgoing link l of g we have to examine whether there are immediate preceding activities within the LCNC of g and s, where s is the target activity of l. The Algorithm 3 $findPrecedings(g,s,lcnc)$ is introduced for that purpose.

The source activities of g's incoming links are per definition the immediate preceding activities of g. Thus, at the beginning of the Algorithm 3 we examine whether g has incoming links.

If g has incoming links, then the source activities of these incoming links are the immediate preceding activities of g (Algorithm 3, line 4-7). In this case, we add tuple (p,k) to *precedings*, where k is an incoming link of g and p is

Algorithm 3: $findPrecedings(g,s,lcnc)$

```
1  begin
2      n = the immediate enclosing activity of g;
3      precedings = ∅ ;
4      if L_g^incoming ≠ ∅ then
5          forall the k ∈ L_g^incoming do
6              p = source activity of k;
7              precedings = precedings ∪ {p};
8      if n is instanceOf sequence then
9          activities = child activities in n;
10         i = index of g in activities;
11         if i ≠ 0 then
12             a_{i-1} = the activity at index i − 1;
13             precedings = precedings ∪ {(a_{i-1}, null)};
14             return precedings;
15         else
16             if n = lcnc then
17                 return null;
18             else
19                 if lcnc = null then
20                     lcnc = findLCNC(g,s);
21                 findPrecedings(n,s,lcnc);
22     continue in Algorithm 4
23 end
```

the source activity of k. The link k in the tuple will be used when removing the outgoing links in Algorithm 6. Otherwise, g has no incoming links and the algorithm continues searching for preceding activities of g.

Depending on the immediate enclosing activity, the algorithm continues with different approaches. We distinguish three kinds of immediate enclosing activity n of g: (i) n is a `<sequence>`, (ii) n is a `<flow>` activity or a `<fragment>` element, (iii) n is an instance of other structured constructs.

Case (i): the activity g is immediately enclosed in a `<sequence>` activity n (Algorithm 3, line 8). According to the index of g in the child activities of n, we can determine whether g has preceding activities.

If the value of g's index i is not 0, then g is not the first activity in n. Thus, the immediate preceding activity of g is then the activity that appears before g in n, i.e. the activity a_i with the index $(i-1)$ (Algorithm 3, line 12). As there exists no link between g and a_i, we add the tuple $(a_i, null)$ to *precedings* (Algorithm 3, line 13). The value *null* indicates that there exists no BPEL link between g and a_i. The algorithm terminates by returning the immediate preceding activities found (Algorithm 3, line 14).

If the value of i equals to 0 (Algorithm 3, line 15), then g is the first activity in n. In this case, the preceding activity could locate outside n. Thus, we have to continue the search until the algorithm reaches the LCNC of g and s.

If the `<sequence>` activity n of g is the LCNC of g and s (Algorithm 3, line 16), then there exists no preceding activities of g in the `<sequence>` activity, as g is the first activity in it. Thus, the algorithm returns *null* (Algorithm 3, line 17).

Otherwise, n is not the LCNC (Algorithm 3, line 18). That means the algorithm has not reached the LCNC of g and s yet. If the LCNC of g and s is *null*, then we have to find the LCNC of g and s (Algorithm 3, line 19-20). After that we invoke the Algorithm 3 recursively to continue the searching (Algorithm 3, line 21).

Case (ii): the activity g is immediately enclosed in a `<flow>` activity or a `<fragment>` element (Algorithm 4, line 2).

If g has incoming links (Algorithm 4, line 4), then the source activities of the incoming links could be the immediate preceding activities of g. Recall that at the beginning of the algorithm (Algorithm 3, line 4-7) we have already added the source activities of g. However, if all the source activities of g's incoming links are located outside n, then g may also have immediate preceding activities outside n but within the LCNC of g and s as shown in Figure 5.19. We can see the activity p_2 is also an immediate preceding activity, which has not been considered in the algorithm yet.

To find p_2 we have to make sure that all source activities of g's incoming links are all located outside n. Otherwise, p_2 is not the immediate preceding activity of g. We use a Boolean variable *isAnEntry* as a flag. At the beginning we assume that g has no immediate preceding activities within n. Thus, we set

Algorithm 4: $findPrecedings(g,s,lcnc)$ continue 1

```
1  begin
2      else if n is instanceOf flow OR fragment then
3          isAnEntry = true;
4          if L_g^incoming ≠ ∅ then
5              while isAnEntry AND L_g^incoming.hasNext do
6                  l = L_g^incoming.next;
7                  a = the source activity of l;
8                  m = immediate enclosing activity of a;
9                  if m = n then
10                     isAnEntry = false;
11             if NOT isAnEntry then
12                 return precedings;
13             else
14                 if n = lcnc then
15                     return precedings;
16                 else
17                     if lcnc = null then
18                         lcnc = findLCNC(g,s);
19                     precedings =
                       precedings ∪ findPrecedings(n,s,lcnc);
20         else
21             if n = lcnc OR n is instanceOf fragment then
22                 return null;
23             else
24                 if lcnc = null then
25                     lcnc = findLCNC(g,s);
26                 findPrecedings(n,s,lcnc);
27     Continue in Algorithm 5
```

the flag $isAnEntry = true$ (Algorithm 4, line 3). We iterate all the incoming links of g, as long as we still have not found an immediate preceding activity of g. If the immediate enclosing activity of the source activity a is n (Algorithm 4, line 7-8), then a is an immediate preceding activity of g in n. Thus, we set the flag $isAnEntry$ to $false$.

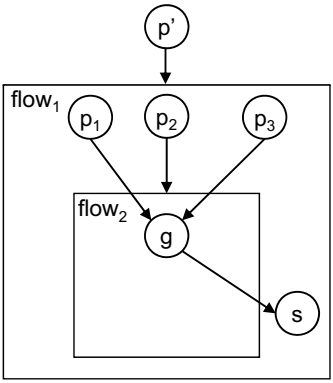


Fig. 5.19 The activity p_2 is also a preceding activity of g within the LCNC of g and s.

If *isAnEntry* is $false$, we found all the immediate preceding activities of g and the algorithm terminates by returning the results (Algorithm 4, line 11-12).

Otherwise, *isAnEntry* is *true* (Algorithm 4, line 13). It means that all source activities of g's incoming links are all located outside n. In this case, there may exist an immediate preceding activity of g, which may have immediate preceding activities outside n but within the LCNC of g and s.

If n is the LCNC of g and s, then there exists no further immediate preceding activities of g within the LCNC of g and s. In this case, we terminates the algorithm by returning the result *precedings* (Algorithm 4, line 14-15). Otherwise, the algorithm continues. If the LCNC of g and s is unknown, then we invoke the Algorithm 2 find the LCNC (Algorithm 4, line 17). After that we invoke the algorithm recursively to continue searching immediate preceding activities of g (Algorithm 4, line 19).

If g has no incoming links (Algorithm 4, line 20), the g may have immediate preceding activities outside n. However, if n is already the LCNC of g and s, then the searching completed. Or if n is root element, then there exists no activities outside n. Thus, the algorithm terminates either (Algorithm 4, line 21-22). Otherwise, the searching continues as discussed before (Algorithm 4, line 23-26).

Case (iii): the activity g is immediately enclosed in a structured construct that is not a `<sequence>`, a `<flow>`, or a `<fragment>` (Algorithm 5, line

Algorithm 5: $findPrecedings(g,s,lcnc)$ continue 2

```
1 begin
2     else
3         if n = lcnc then
4             return null;
5         else
6             if lcnc = null then
7                 lcnc = findLCNC(g,s);
8             findPrecedings(n,s,lcnc);
9 end
```

2). For example, *n* could be `<if>`, `<else>`, `<while>`, `<scope>`, etc. These constructs allow only enclosing exactly one activity. This means that the immediate preceding activity of g must be located outside n. Thus, we invoke the algorithm recursively to continue searching as discussed before (Algorithm 5, line 3-8).

5.4.2.2 Clean Outgoing Links

Now we continue the discussion of the Algorithm 6 $cleanOutgoingLinks(g)$.

We iterate the outgoing links of g (Algorithm 6, line 3). We examine whether there are preceding activities of g within the LCNC of g and s (Algorithm 6, line 5), where s denotes the target activity of the outgoing link l.

If g has preceding activities within the LCNC of g and s (Algorithm 6, line 6), then we first remove the link l (Algorithm 6, line 7).

If s is not reachable from p any more (Algorithm 6, line 9), then there exists no alternative path from p to s besides the path over the link l. According to Rule 4 we create a new link (p,s) to restore the original control dependency between p and s (Algorithm 6, line 10).

If s is still reachable after removing l, then there exists an alternative path from p to s that not contains l. The original control dependency between p and s is retained through the alternative path. Thus, according to Rule 3 the

Algorithm 6: *cleanOutgoingLinks*(g)

```
1  begin
2      u = null;
3      forall the l ∈ L_g^outgoing do
4          s = target activity of l;
5          precedings = findPrecedings(g, s, null);
6          if precedings ≠ ∅ then
7              remove l;
8              forall the p ∈ precedings do
9                  if s is not reachable from p then
10                     create the link (p, s);
11         else
12             remove l;
13 end
```

algorithm does nothing, as the link l has already been removed (Algorithm 6, line 7).

If g has no preceding activities within the LCNC of g and s (Algorithm 6, line 11), then according to Rule 2 the algorithm removes l.

5.4.2.3 Find Succeeding Activities

Analogously, the Algorithm 10 *cleanIncomingLinks*(g) uses the Algorithm 7 $findSucceedings$ to discover the immediate succeeding activities of the generated opaque activity g. As the algorithms work in the similar manner as the algorithms for removing outgoing links of an opaque activity, we do not dis-

cuss them here in details. Here we use p to denote the source activity of the incoming link of g that will be removed through the algorithm below.

Algorithm 7: $findSucceedings(g, p, lcnc)$

```
1  begin
2      n = the immediate enclosing activity of g;
3      succeedings = ∅ ;
4      if L_g^outgoing ≠ ∅ then
5          forall the k ∈ L_g^outgoing do
6              s = target activity of k;
7              succeedings = succeedings ∪ {s};
8      if n is instanceOf sequence then
9          activities = child activities in n;
10         i = index of g in activities;
11         k = the number of activities in n;
12         if i ≠ k − 1 then
13             a_{i+1} = the activity at index i + 1;
14             add (a_{i+1}, null) to succeedings;
15             return succeedings;
16         else
17             if n = lcnc then
18                 return null;
19             else
20                 if lcnc = null then
21                     lcnc = findLCNC(g, p);
22                 findSucceedings(n, p, lcnc);
23     Continue in Algorithm 8
```

Algorithm 8: $findSucceedings(g, p, lcnc)$ continue 1

```
1  begin
2      else if n is instanceOf flow OR fragment then
3          isAnExit = true;
4          if L_g^outgoing ≠ ∅ then
5              while isAnExit AND L_g^outgoing.hasNext do
6                  l = L_g^outgoing.next;
7                  a = the target activity of l;
8                  m = immediate enclosing activity of a;
9                  if m = n then
10                     isAnExit = false;
11             if NOT isAnExit then
12                 return succeedings;
13             else
14                 if n = lcnc then
15                     return succeedings;
16                 else
17                     if lcnc = null then
18                         lcnc = findLCNC(g, p);
19                     succeedings =
                       succeedings ∪ findSucceedings(n, p, lcnc);
20         else
21             if n = lcnc OR n is instanceOf fragment then
22                 return null;
23             else
24                 if lcnc = null then
25                     lcnc = findLCNC(g, p);
26                 findSucceedings(n, p, lcnc);
27 end
```

Algorithm 9: $findSucceedings(g, p, lcnc)$ continue 2

```
1  begin
2      else
3          if n = lcnc then
4              return null;
5          else
6              if lcnc = null then
7                  lcnc = findLCNC(g, p);
8              findSucceedings(n, p, lcnc);
9  end
```

Algorithm 10: $cleanIncomingLinks(g)$

```
1  begin
2      forall the l ∈ L_g^incoming do
3          p = source activity of l;
4          succeedings = findSucceedings(g, p, null);
5          if succeedings ≠ ∅ then
6              remove l;
7              forall the s ∈ succeedings do
8                  if s is not reachable from p then
9                      create the link (p, s);
10         else
11             remove l;
12 end
```

5.4.2.4 Clean Fragment, Flow, and Sequence Constructs

The modeling constructs `<fragment>`, `<flow>`, and `<sequence>` share the same procedure for removing generated opaque activities. Thus, we introduce the Algorithm 11 $clean(construct)$ for cleaning the generated opaque activities from these constructs.

Algorithm 11 examines each activity *a* that is immediately enclosed in *construct* (Algorithm 11, line 2). The basic idea is still first to remove all the incoming and outgoing links of a generated opaque activity, then to remove the generated opaque activity itself.

Algorithm 11: *clean*(*construct*)

```
1  begin
2      forall the immediate enclosed activity a in construct do
3          if a is a generated opaque then
4              if a has incoming links then
5                  cleanIncomingLinks(a);
6              if a has outgoing links then
7                  cleanOutgoingLinks(a);
8              remove a;
9          else if a is not a basic activity then
10             cleanConstruct(a);
11 end
```

If the activity *a* is a generated opaque activity (Algorithm 11, line 3), then the algorithm examines which reduction rules can be applied to remove it.

If *a* has incoming links, then we use the Algorithm 10 *cleanIncomingLinks* to removing them (Algorithm 11, line 4-5). If *a* has outgoing links, then the algorithm uses the Algorithm 6 *cleanOutgoingLinks* to removing them (Algorithm 11, line 6-7).

If *a* is not a basic activity, then we invoke the Algorithm 12 to process the structured activities (Algorithm 11, line 9-10).

Algorithm 12: *cleanConstruct*(a)

```
1  begin
2      switch a do
3          case fragment
4              clean(construct);
5          case flow
6              clean(construct);
7          case forEach
8              cleanLoop(a);
9          case if
10             cleanIf(a);
11         case pick
12             cleanPick(a);
13         case repeatUntil
14             cleanLoop(a);
15         case sequence
16             clean(construct);
17         case scope
18             cleanScope(a);
19         case while
20             cleanLoop(a);
21         case compensationHandler
22             cleanCompensationHandler(a);
23         case eventHandlers
24             cleanEventHandlers(a);
25         case faultHandlers
26             cleanFaultHandlers(a);
27         case terminationHandler
28             cleanTerminationHandler(a);
29 end
```

5.4.2.5 Clean If Activity

An <if> activity may contain more than one branches, each of which allows only exactly one primary activity (let's call it a). For each branch we follow the same logic for cleaning generated opaque activities as the one described for cleaning <fragment>, <flow>, and <sequence> constructs.

Algorithm 13: $cleanIf(if)$

```
1  begin
2      a = the primary activity of if;
3      if a is a generated opaque activity then
4          cleanIncomingLinks(a);
5          cleanOutgoingLinks(a);
6          remove a;
7      else if a is not a basic activity then
8          cleanConstruct(a);
9      forall the elseIf in if do
10         a = the primary activity of elseIf;
11         if a is a generated opaque activity then
12             cleanIncomingLinks(a);
13             cleanOutgoingLinks(a);
14             remove a;
15         else if a is not a basic activity then
16             cleanConstruct(a);
17     if if has else branch then
18         a = the primary activity of else;
19         if a is a generated opaque activity then
20             cleanIncomingLinks(a);
21             cleanOutgoingLinks(a);
22             remove a;
23         else if a is not a basic activity then
24             cleanConstruct(a);
25 end
```

5.4.2.6 Clean Pick Activity

A <pick> activity contains at least one <onMessage> construct and may contain one or more <onAlarm> constructs, each of which contains exactly one activity. To clean a <onMessage> and the <onAlarm> we also use the same procedures as discussed before.

Algorithm 14: *cleanPick*(*pick*)

```
1  begin
2      forall the onMessage in pick do
3          a = the primary activity of onMessage;
4          if a is a generated opaque activity then
5              cleanIncomingLinks(a);
6              cleanOutgoingLinks(a);
7              remove a;
8          else if a is not a basic activity then
9              cleanConstruct(a);
10     forall the onAlarm in pick do
11         a = the primary activity of onAlarm;
12         if a is a generated opaque activity then
13             cleanIncomingLinks(a);
14             cleanOutgoingLinks(a);
15             remove a;
16         else if a is not a basic activity then
17             cleanConstruct(a);
18 end
```

5.4.2.7 Clean Scope Activity

A <scope> activity allows to immediately enclose exactly one primary activity. If the primary activity *a* is a generated opaque activity, then we use the same procedure to remove it as discussed before (Algorithm 15, line 3-8).

A <scope> activity may also contain compensation, fault, event, and termination handlers. The algorithm examines whether such handlers are immediately used in the <scope> activity and apply the corresponding algorithm for further processing. We will discuss these algorithms later in details.

Algorithm 15: *cleanScope*(*scope*)

```
1  begin
2      a = the primary activity of scope;
3      if a is a generated opaque activity then
4          cleanIncomingLinks(a);
5          cleanOutgoingLinks(a);
6          remove a;
7      else if a is not a basic activity then
8          cleanConstruct(a);
9      if scope contains compensation handler ch then
10         cleanCompensationHandler(ch);
11     if scope contains event handlers eh then
12         cleanEventHandlers(eh);
13     if scope contains fault handlers fh then
14         cleanFaultHandlers(fh);
15     if scope contains terminate handlers th then
16         cleanTerminationHandler(th);
17 end
```

5.4.2.8 Clean Compensation Handler

A <compensationHandler> allows also only a single primary activity. However, a link must not cross the boundary of a <compensationHandler> [16]. Thus, if the primary activity *a* of a <compensationHandler> is a generated opaque activity, then it has neither incoming nor outgoing links and is an isolated opaque activity. Therefore, the algorithm can simply remove it (Rule 1) (line 3-4).

Algorithm 16: *cleanCompensationHandler*(*ch*)

```
1 begin
2 |  a = the primary activity of ch;
3 |  if a is a generated opaque activity then
4 |  |  remove a;
5 |  else if a is not a basic activity then
6 |  |  cleanConstruct(a);
7 end
```

5.4.2.9 Clean Event Handlers

The algorithm for cleaning `<eventHandler>` works in an analogous manner as that for a `<pick>` activity. Here we know in advance that the primary activity of each `<onEvent>` and `<onAlarm>` construct is a `<scope>` activity. Thus, the algorithm can directly invoke the algorithm *cleanScope* for further processing.

Algorithm 17: *cleanEventHandlers*(*eh*)

```
1 begin
2 |  forall the onEvent in eh do
3 |  |  a = the primary activity in onEvent;
4 |  |  cleanScope(a);
5 |  forall the onAlarm in pick do
6 |  |  a = the primary activity in onAlarm;
7 |  |  cleanScope(a);
8 end
```

5.4.2.10 Clean Fault Handlers

The `<faultHandlers>` construct may contain one or more `<catch>` constructs and one `<catchAll>` construct. As the link that crosses the boundary of a `<catch>` or a `<catchAll>` construct must be a outgoing link [16],

they are not allowed to have incoming links. In case the primary activity of a <catch> or a <catchAll> construct is a generated opaque activity, we only have to examine whether it has outgoing links. In case there are none, the generated opaque activity is then an isolated opaque activity. Thus, the algorithm can simply remove it (Rule 1).

Algorithm 18: *cleanFaultHandlers*(fh)

```
1  begin
2      forall the catch in fh do
3          a = the primary activity of catch;
4          if a is a generated opaque activity then
5              if L_a^outgoing = ∅ then
6                  remove a;
7              else
8                  cleanOutgoingLinks(a);
9                  remove a;
10         else if a is not a basic activity then
11             cleanConstruct(a);
12     if fh has catchAll then
13         a = the primary activity of catchAll;
14         if a is a generated opaque activity then
15             if L_a^outgoing = ∅ then
16                 remove a;
17             else
18                 cleanOutgoingLinks(a);
19                 remove a;
20         else if a is not a basic activity then
21             cleanConstruct(a);
22 end
```

5.4.2.11 Clean Termination Handler

The <terminationHandler> allows also exactly one primary activity. And a link that crosses the boundary of <terminationHandler> must

be a outgoing link. Thus, if the primary activity a does not have outgoing links, then it is an isolated opaque activity.

Algorithm 19: *cleanTerminationHandler*(th)

```
1  begin
2      a = the primary activity of th;
3      if a is a generated opaque activity then
4          if L_a^outgoing = ∅ then
5              remove a;
6          else
7              cleanOutgoingLinks(a);
8              remove a;
9      else if a is not a basic activity then
10         cleanConstruct(a);
11 end
```

5.4.2.12 Clean Repeatable Construct

The Algorithm 20 *cleanLoopConstruct* removes generated opaque activities in all repeatable constructs of BPEL, i.e. `<while>`, `<repeatUntil>`, and `<forEach>`. As specified in the BPEL standard, a link must not cross the boundary of a repeatable construct. Thus, if the primary activity in the repeatable construct is a generated opaque activity, then it is an isolated opaque activity. Thus, it can be simply removed. Otherwise, the algorithm invokes the Algorithm 12 *cleanConstruct* to apply the appropriate algorithm for further processing.

5.4.2.13 Clean Empty Complex Constructs

Some complex constructs may contain only generated opaque activities. After removing the generated opaque activities, the complex construct becomes an empty construct. If the complex construct itself has not been selected in the

Algorithm 20: *cleanLoopConstruct*(*loop*)

```
1 begin
2 |  a = the primary activity of loop;
3 |  if a is a generated opaque activity then
4 |  |  remove a;
5 |  else if a is not a basic activity then
6 |  |  cleanConstruct(a);
7 end
```

selection phase, then it should also be removed to keep the resulting BPEL fragment compact.

The Algorithm 21 utilizes the Nesting Hierarchy Tree (NHT) (Algorithm 21, line 2) to clean empty complex constructs. It traverses the NHT in a bottom-up approach (Algorithm 21, line 3), i.e. from the leaves to the root. If a leaf node in the NHT contains no basic activities and is not in the original selection of the process modeler, then the algorithm removes the leaf node from the fragment (Algorithm 21, line 4-7).

Algorithm 21: *cleanEmptyComplexConstruct*(*fragment*, *selection*)

```
1 begin
2 |  nht = createNHT(fragment);
3 |  nodes[] = a sorted array of the nodes in nht in descending order of
  |  their level;
4 |  for i = 0 to nodes[].size − 1 do
5 |  |  if nodes[i] is empty then
6 |  |  |  if nodes[i] ∉ selection then
7 |  |  |  |  remove nodes[i];
8 end
```

Chapter 6
Mapping BPEL Process Models to Graph

This chapter provides a mathematical framework for mapping a BPEL process or a BPEL fragment to a Directed and Acyclic Graph (DAG). In Section 6.1 we outline the needs of extending the graph-based meta-model proposed by Khalaf [78]. Section 6.2 describe the common mapping rules that apply for the whole mapping framework. In Section 6.3 we define the rules for mapping basic activities in BPEL to a graph representation. Section 6.4 presents the rules for mapping structured activities (except for a scope activity) in BPEL to a graph representation. Due to the specialty of the scope activity in BPEL we introduce the rules for mapping BPEL scopes and its immediately enclosed handlers in Section 6.5.

6.1 Introduction

In a graph-based approach a process model can be represented as a rooted, Directed, and Acyclic Graph (DAG), in which nodes represent activities in the given process model and directed edges represent control flow of the activities. Leymann et al. [95] developed a graph-based meta-model. The meta-model provides the syntactical constructs for users to create workflow models. In addition, it also defines the operational semantics so that process engines are able to execute process models in a consistent way. Khalaf [78] extended the meta-

model to adapt to specific characteristics of BPEL process models, such as loops, nested scopes, fault handling, etc. Khalaf also defined a mapping framework for transforming BPEL process models to a graph-based formalism. The mapping framework lays groundwork for our graph mapping framework for BPEL process models and fragments, but does not meet all the requirements that we have identified in our research.

In the mapping framework of Khalaf, a BPEL scope is mapped to a hyperedge of a hypergraph [29]. As a hyperedge can connect any number of nodes, the parent-child relationships of the nodes does not reflect exactly the original control dependencies. The algorithms we present in this thesis need a graph representation that captures the exact process structure of the BPEL process or fragment. Process structure refers to the way in which process activities are connected. It does not consider the operational semantics of the process model. It is a static representation of the construction of the BPEL process or fragment.

In addition, the mapping framework in [78] connects a flow activity with all its enclosed activities. The redundant edges may distort the original control dependencies of the activities. The query algorithm in this thesis processes the structural matchmaking by examining the parent-child relationships of the matching nodes. For that reason the edges in the graph representation of BPEL processes or fragments should explicitly reflect the original parent-child relationships between the process activities.

Last but not least, in the mapping framework of Kahlaf, an invoke activity is mapped to exactly one node, even though the invoke activity may contain compensation and fault handlers. Semantically, an invoke activity with enclosed compensation and fault handler is equivalent to a scope activity that immediately encloses the invoke activity as well as the compensation and fault handlers. To enable the matchmaking of such invoke activities with scope activities, we need to transform the invoke activity to a scope activity and map the resulting scope activity to the graph representation.

In the following we introduce our mapping framework that will be used in Chapter 7 for query processing. We use $G = (V, E)$ to denote the graph representation of a BPEL process or fragment, where V is the set of nodes and E is the set of edges.

6.2 Common Mapping Rules

The graph-based meta-model is aiming to provide a formalized representation of the process structures of a BPEL process or fragment. As we discussed in the beginning of this chapter, the structure refers to the way in which process activities are connected together. It focuses on the static snapshot of the construction of BPEL processes and fragments at design time rather than their operational semantics at runtime. Transition conditions, join conditions, partner links, variables, correlation sets, and message exchanges will not be mapped to the graph representation. However, in the prototypical implementation we use this data for keyword-based query processing, which provides users, in combination with the structural matchmaking, a more powerful query capability.

Attributes of process activities can be used as predicates for matchmaking of activity nodes in our query algorithm. As a general rule, we use t_x as the unique identifier of a node in the graph representation. The letter t denotes the type of the BPEL activity, such as invoke, flow, scope, etc. The subscript x indicates different instances of the same activity type in the graph. All the other attributes of an activity are mapped as labels of the respective node.

We use V to denote the set of nodes and E to denote the set of edges in the resulting graph. The function $\gamma : A \rightarrow P(V)$ maps a set of BPEL activities A to a power set ($P(V)$) of nodes in the process graph. The function $\theta : A \rightarrow P(E)$ maps a set of BPEL activities A to a power set of ($P(V)$) of edges in the process graph.

The function γ is defined as follows:

$$\begin{cases} \gamma(a) = \{v\} & a \in A \text{ is basic activity} \\ \gamma(a) = \{\alpha_a, \beta_a\} & a \in A \text{ is not a basic activity} \end{cases}$$

The function θ is defined as follows:

$$\begin{cases} \theta(a) = \emptyset & a \in A \text{ is basic activity} \\ \theta(a) = \{\alpha_a, \beta_a\} & a \in A \text{ is not a basic activity} \end{cases}$$

In our mapping framework we do not address the mapping rules for incomplete links. We leave it as future work.

6.3 Mapping Basic Activities

A basic activity is mapped to a single node $v \in V$ in the process graph. BPEL allows users to define *extension activities*. The graph mapping framework also maps an extension activity to a single node in the graph. However, the implementation of the mapping framework is designed to be extensible, so that users can define domain specific mappings for extension activities.

We use a to denote an activity. The function $\gamma : A \rightarrow P(V)$ maps a set of BPEL activities A to a power set ($P(V)$) of nodes in the process graph. The function γ is defined as follows:

$$\gamma(a) = \{v\}, \quad a \in A \text{ is basic activity}$$

A special case occurs when mapping an `<invoke>` activity with enclosed compensation and fault handlers. Such an `<invoke>` activity is semantically equivalent to a `<scope>` activity that immediately encloses the `<invoke>` activity as well as the compensation and fault handlers [16]. In order to provide a consistent graph-based representation on the structure of BPEL process models and fragments, we transform each `<invoke>` activity with enclosed compensation and fault handlers in the following steps:

1. Create a new and empty `<scope>`;
2. Move the enclosed elements of the `<invoke>` activity, such as correlations, sources, targets, the compensation handler, and fault handlers into the `<scope>` activity;
3. Set the `<scope>` activity as the target activity of all the incoming links of the `<invoke>` activity;
4. Set the `<scope>` activity as the source activity of all the outgoing links of the `<invoke>` activity;

5. Replace <fromParts> and <toParts> with <assign> activities as specified in BPEL Standard [16];
6. Add the <invoke> activity as the primary activity of the target <scope> activity.

The resulting <scope> activity is then be mapped to the graph according to the rules specified in Section 6.5.

6.4 Mapping Structured Activities

A structured activity is mapped to two nodes in the graph, i.e. one *start node* and one *end node*. The start and end nodes are used to indicate the boundary of the structured activity.

6.4.1 Mapping Sequential Processing - Sequence

A <sequence> activity contains one or more activities that should be executed sequentially. The order of their appearance indicates the sequential order of their execution. We denote a sequence activity $A_{sequence}$ as a set of BPEL activities, i.e.

$$A_{sequence} = \{a_1, a_2, ..., a_n\}$$

A <sequence> activity itself is mapped to a start node $\alpha_{sequence}$ and an end node $\beta_{sequence}$. All its immediately enclosed activities $\{a_1, a_2, ..., a_n\}$ in the <sequence> are mapped recursively according to the mapping rules specified in this chapter. Thus, the set of nodes $V_{sequence}$ resulted by mapping the <sequence> activity is defined as follows:

$$V_{sequence} = \gamma(sequence) = \{\alpha_{sequence}, \beta_{sequence}\} \cup \bigcup_{i=1}^{n} \gamma(a_i) \tag{6.4.1}$$

The mapping function $\theta sequence$ creates edges to connect the transformed nodes together. The execution semantic of a <sequence> activity is that the first activity a_1 may not start, until the <sequence> itself has started. And the <sequence> activity completes in the normal mode when the last activity a_n in the <sequence> activity has completed. Each activity a_{i+1} may not start until its preceding activity a_i has completed. Therefore, we create a directed edge from the start node of the sequence $\alpha_{sequence}$ to the start node of the graph representation of the first activity $\alpha(\gamma(a_1))$. For the immediately enclosed activities in the <sequence> activity we connect the end node of the preceding activity $\beta(\gamma(a_i))$ with the start node of the following activity $\alpha(\gamma(a_{i+1}))$. At the end, an edge is created from the end node of the last immediately enclosed activity in the <sequence> activity $\beta(\gamma(a_n))$ to the end node of the <sequence> activity $\beta_{sequence}$.

The mapping function $\theta(sequence)$ is defined as follows:

$$\begin{aligned} \theta(sequence) = & \{(\alpha_{sequence}, \alpha(\gamma(a_1))), true\} \cup \\ & \bigcup_{i=1}^{n-1} \{\beta(\gamma(a_i)), \alpha(\gamma(a_{i+1}))), true\} \cup \\ & \{(\beta(\gamma(a_n)), \beta_{sequence}), true\} \end{aligned} \tag{6.4.2}$$

Thus, the set of edges $E_{sequence}$ resulted by mapping the <sequence> activity is the union of $\theta(sequence)$ with the edges created by mapping each enclosed activity in the <sequence> activity.

$$E_{sequence} = \theta(sequence) \cup \bigcup_{i=1}^{n} \theta(a_i) \tag{6.4.3}$$

6.4.2 Mapping Parallel Processing - Flow

A <flow> activity comprises a set of activities and a set of links, which specifies the control dependencies between its enclosed activities. We denote a <flow> activity as a tuple (A_{flow}, L_{flow}). Note that the L_{flow} denotes the <link>s that are immediately used within the <flow> activity.

$$A_{flow} = \{a_1, a_2, ..., a_n\}$$
$$L_{flow} = \{l_1, l_2, ..., l_m\}$$

Analogous to the mapping of a <sequence> activity, we also map a <flow> activity itself to a start node and an end node. In addition, we also map the immediately enclosed activities according to the mapping rules defined in this chapter. Thus, the set of nodes V_{flow} resulted by mapping the <flow> activity is defined as follows:

$$V_{flow} = \gamma(flow) = \{\alpha_{flow}, \beta_{flow}\} \cup \bigcup_{i=1}^{n} \gamma(a_i) \tag{6.4.4}$$

A fundamental semantic of a flow activity is to enable modeling concurrency of a group of activities. None of the enclosed activities in a flow can be activated unless the flow activity itself has been activated. A group of immediately enclosed activities in a flow activity with no incoming links that are defined in the flow activity are called *candidate* start activities in this thesis. These activities may have incoming links that are defined in an enclosing flow activity. In this case, the start of the activities does not depend solely on the activation state of the immediate enclosing flow activity but depends also on the evaluation of the transition condition of the incoming link.

A flow activity completes when all enclosed activities in the flow activity have been completed or reached the state *dead* during the dead path elimination. Immediately enclosed activities in the flow activity with no outgoing links that are defined in the flow activity are called end activities. Thus, the set of start activities A_{start} and the set of end activities A_{end} are defined as follows:

$$A_{start} = \{a_i | (L_{a_i}^{incoming} = \emptyset) \vee (\vee_i (a_i = \pi_2(l) \wedge \pi_1(l) \notin A_{flow})), l \in L_{a_i}^{incoming}\}$$
$$A_{end} = \{a_i | (L_{a_i}^{outgoing} = \emptyset) \vee (\vee_i (a_i = \pi_1(l) \wedge \pi_2(l) \notin A_{flow})), l \in L_{a_i}^{outgoing}\}$$
$$\setminus A_{start}$$

The mapping function $\theta(flow)$ for mapping the edges is defined as follows. In the formula below we use the function $\phi(l_j)$ to denote the transition condition that is associated with the link l_j.

$$\begin{aligned}\theta(flow) = & \bigcup_{a_i \in A_{start}} \{(\alpha_{flow}, \alpha(\gamma(a_i))), true\} \cup \\ & \bigcup_{j=1}^{m} \{(\beta(\gamma(\pi_1(l_j)))), \alpha(\gamma(\pi_2(l_j))), \phi(l_j)\} \cup \\ & \bigcup_{a_k \in A_{end}} \{(\beta(\gamma(a_k)), \beta flow), true\}\end{aligned} \tag{6.4.5}$$

The set of edges E_{flow} resulted by mapping the `<flow>` activity is the union of $\theta(flow)$ with the edges created by mapping each enclosed activity in the `<flow>` activity.

$$E_{flow} = \theta(flow) \cup \bigcup_{i=1}^{n} \theta(a_i) \tag{6.4.6}$$

6.4.3 Mapping Conditional Behavior - If

An `<if>` activity comprises of one or more conditional branches and an optional branch, each of which contains an activity. As only one branch can be performed, the first branch that satisfies the condition is taken and its contained activity is executed. If no conditional branch is taken and a default branch is present, then the default branch is performed. An `<if>` activity completes, when (i) the contained activity in the selected branch completes; (ii) no con-

dition evaluates to true and no default branch is present. We denote an <if> activity as a set of $(condition, activity)$ tuples, where c_i denotes the condition and a_i denotes the activity.

$$A_{if} = \{(c_1, a_1), (c_2, a_2), ..., (c_n, a_n))\} \tag{6.4.7}$$

In an <if> activity, the elements <elseIf> and <else> are only used as wrappers to structure conditional branches. We do not map <elseIf> and <else> to the graph, but link the activity in each branch with the start node of the <if> activity and bind the respective condition with the corresponding directed edge. To do so we can avoid redundant start and end nodes of the conditional branches and makes the graph more compact. Thus, the set of nodes V_{if} resulted by mapping the <if> activity is defined as follows:

$$V_{if} = \gamma(if) = \{\alpha_{if}, \beta_{if}\} \cup \bigcup_{i=1}^{n} \gamma(a_i) \tag{6.4.8}$$

The mapping function $\theta(if)$ is defined as follows:

$$\theta(if) = \bigcup_{i=1}^{n} \{(\alpha_{if}, \alpha(\gamma(a_i)), c_i)\} \cup \bigcup_{j=1}^{n} \{(\beta(\gamma(a_j)), \beta_{if}), true\} \tag{6.4.9}$$

The set of edges E_{if} resulted by mapping the <if> activity is the union of $\theta(if)$ with the edges created by mapping the enclosed activity in each branch.

$$E_{if} = \theta(if) \cup \bigcup_{i=1}^{n} \theta(a_i) \tag{6.4.10}$$

6.4.4 Mapping Repetitive Execution - While

A <while> activity is used to specify repeated execution of a contained activity. The contained activity is executed as long as the loop condition evaluates to true the beginning of each iteration. We denote a <while> activity as a

$(condition, activity)$ tuple, where c denotes the condition and a denotes the activity.

$$A_{while} = \{(c,a)\}$$

Thus, the set of nodes V_{while} resulted by mapping the <while> activity is defined as follows:

$$V_{while} = \gamma(while) = \{\alpha_{while}, \beta_{while}\} \cup \gamma(a) \tag{6.4.11}$$

The mapping function $\theta(while)$ is defined as follows:

$$\theta(while) = \{(\alpha_{while}, \alpha(\gamma(a)), c)\} \cup \{(\beta(\gamma(a)), \beta_{while}, true)\} \tag{6.4.12}$$

The set of edges E_{while} resulted by mapping the <while> activity is the union of $\theta(while)$ with the edges created by mapping the enclosed activity.

$$E_{while} = \theta(while) \cup \theta(a) \tag{6.4.13}$$

6.4.5 Mapping Repetitive Execution - RepeatUntil

A <repeatUntil> activity provides another possibility to define repetitive execution. Different to the <while> activity, the contained activity in a <repeatUntil> activity is executed at least once. The condition is evaluated after each execution of the contained activity and the <repeatUntil> activity completes when the condition evaluates to true. We denote a <repeatUntil> activity as a $(condition, activity)$ tuple, where c denotes the condition and a denotes the activity.

$$A_{repeatUntil} = \{(c,a)\}$$

Thus, the set of nodes $V_{repeatUntil}$ resulted by mapping the <repeatUntil> activity is defined as follows:

$$V_{repeatUntil} = \gamma(repeatUntil) = \{\alpha_{repeatUntil}, \beta_{repeatUntil}\} \cup \gamma(a) \quad (6.4.14)$$

Likewise, the mapping of a <repeatUnitl> activity generates a set of nodes including the start node and the end node for the <repeatUnitl> activity and all the nodes created by mapping the activity a enclosed in it. Unlike the mapping of the <while> activity we bound the iteration condition of the <repeatUnitl> activity with the edge that linking the end node of the enclosing activity and the end node of the <repeatUnitl> activity. This transformation reflects the fact that the enclosing activity is executed at least once.

The mapping function $\theta(repeatUntil)$ is defined as follows:

$$\begin{aligned}\theta(repeatUntil) = &\{(\alpha_{repeatUntil}, \alpha(\gamma(a)), true)\} \cup \\ &\{(\beta(\gamma(a)), \beta_{repeatUntil}, c)\}\end{aligned} \quad (6.4.15)$$

The set of edges $E_{repeatUntil}$ resulted by mapping the <repeatUntil> activity is the union of $\theta(repeatUntil)$ with the edges created by mapping the enclosed activity.

$$E_{repeatUntil} = \theta(repeatUntil) \cup \theta(a) \quad (6.4.16)$$

6.4.6 Mapping Selective Event Processing - Pick

A <pick> activity allows a process to react to the occurrence of exactly one event among a set of pre-defined events. It comprises a set of branches, each of which is a event-activity pair. The activity associated with the selected event will be executed and all the other events are ignored by the <pick> activity. A <pick> activity completes when the selected activity completes. As BPEL provides two types of events, i.e. the <onMessage> event and the time-based <onAlarm> event, we denote a <pick> activity as a set of $(event, activity)$ tuples.

$$\begin{aligned}A_{pick} =& A_{message} \cup A_{alarm} \\ =& \left\{(o_{message_1}, a_1), (o_{message_2}, a_2), ..., (o_{message_m}, a_m)\right\} \cup \\ & \left\{(o_{alarm_1}, a_{m+1}), (o_{alarm_2}, a_{m+2}), ..., (o_{alarm_n}, a_{m+n})\right\} \\ & 1 \leq m \text{ and } 1 \leq n\end{aligned}$$

Thus, the set of nodes V_{pick} resulted by mapping the `<pick>` activity is defined as follows:

$$\begin{aligned}V_{pick} = \gamma(pick) = \left\{\alpha_{pick}, \beta_{pick}\right\} \cup \bigcup_{i=1}^{m} \left\{\alpha_{o_{message_i}}, \beta_{o_{message_i}}\right\} \cup \\ \bigcup_{j=1}^{n} \left\{\alpha_{o_{alarm_j}}, \beta_{o_{alarm_j}}\right\} \cup \bigcup_{k=m+1}^{m+n} \gamma(a_k)\end{aligned} \tag{6.4.17}$$

The mapping function $\theta(pick)$ is defined as follows:

$$\begin{aligned}\theta(pick) = & \bigcup_{i=1}^{m} \left\{(\alpha_{pick}, \alpha(o_{message_i}), true)\right\} \cup \\ & \bigcup_{j=1}^{n} \left\{(\alpha_{pick}, \alpha(o_{alarm_j}), true)\right\} \cup \\ & \bigcup_{i=1}^{m} \left\{(\alpha(o_{message_i}), \alpha(\gamma(a_i)), true)\right\} \cup \\ & \bigcup_{j=1}^{n} \left\{(\alpha(o_{alarm_j}), \alpha(\gamma(a_{m+j})), true)\right\} \cup \\ & \bigcup_{i=1}^{m} \left\{(\beta(\gamma(a_i)), \beta(o_{message_i}), true)\right\} \cup \\ & \bigcup_{j=1}^{n} \left\{(\beta(\gamma(a_{m+j})), \beta(o_{alarm_j}), true)\right\} \cup \\ & \bigcup_{i=1}^{m} \left\{(\beta(o_{message_i}), \beta_{pick}, true)\right\} \cup \\ & \bigcup_{j=1}^{n} \left\{(\beta(o_{alarm_j}), \beta_{pick}, true)\right\}\end{aligned} \tag{6.4.18}$$

The set of edges E_{pick} resulted by mapping the <pick> activity is the union of $\theta(pick)$ with the edges created by mapping the enclosed activity of each branch.

$$E_{pick} = \theta(pick) \cup \bigcup_{k=1}^{m+n} \theta(a) \tag{6.4.19}$$

6.4.7 Processing Multiple Branches - ForEach

A <forEach> activity allows to execute the enclosed <scope> activity exactly $N+1$ times, where N is the difference between the values of the start and the final counter. The enclosed activity can be repeated in a serial or a parallel manner. In case of parallelism parallel branches will be dynamically generated, whose number is not known at design time. However, our mapping framework considers only the static process structure but no operational semantics. Thus, both serial and parallel <forEach> activities will be mapped according to its static process structure defined at design time. We denote a <forEach> activity as a tuple of $(condition, scope)$, where c denotes the condition and *scope* denotes the <scope> activity.

$$A_{forEach} = \{(c, scope)\}$$

Thus, the set of nodes $V_{forEach}$ resulted by mapping the <forEach> activity is defined as follows:

$$V_{forEach} = \gamma(forEach) = \{\alpha_{forEach}, \beta_{forEach}\} \cup \gamma(scope) \tag{6.4.20}$$

The mapping function $\theta(forEach)$ is defined as follows:

$$\begin{aligned} \theta(forEach) = &\{(\alpha_{forEach}, \alpha(\gamma(scope)), c)\} \cup \\ &\{(\beta(\gamma(scope)), \beta_{forEach}, true)\} \end{aligned} \tag{6.4.21}$$

The set of edges $E_{forEach}$ resulted by mapping the <forEach> activity is the union of $\theta(forEach)$ with the edges created by mapping the enclosed activity.

$$E_{forEach} = \theta(forEach) \cup \theta(scope) \tag{6.4.22}$$

6.5 Mapping BPEL Scope

A <scope> activity defines a local context which influences the behavior of its enclosed activities at run time. The local context may include variables, partner links, message exchanges, correlation sets, event handlers, fault handlers, a compensation handler, and a termination handler. As our mapping framework focuses on the static process structure of a BPEL process or fragment, it does not map the variables, partner links, message exchanges, and correlation sets to the graph representation. However, handlers will be mapped to the graph representation. We consider a <scope> as tuple of $(A_{scope}, A_{handlers})$. In the following we use *ch* to denote compensation handler, *eh* to denote event handlers, fh to denote fault handlers, and th to denote termination handler.

$$A_{scope} = \{a\}$$
$$A_{handlers} = \{ch, eh, fh, th\}$$

Thus, the set of nodes V_{scope} resulted by mapping the <scope> activity is defined as follows:

$$\begin{aligned} V_{scope} =& \gamma(scope) \cup \gamma(handlers) \\ =& \{\alpha_{scope}, \beta_{scope}\} \cup \gamma(a) \cup \\ & \gamma(ch) \cup \gamma(eh) \cup \gamma(fh) \cup \gamma(th) \end{aligned} \tag{6.5.1}$$

The mapping function $\theta(scope)$ is defined as follows:

$$\begin{aligned}\theta(scope) = &\{(\alpha_{scope}, \alpha(\gamma(a))), true\} \cup \\ &\{(\alpha_{scope}, \alpha(\gamma(ch))), undefined\} \cup \\ &\{(\alpha_{scope}, \alpha(\gamma(eh))), true\} \cup \\ &\{(\alpha_{scope}, \alpha(\gamma(fh))), true\} \cup \\ &\{(\alpha_{scope}, \alpha(\gamma(th))), true\} \cup \\ &\{(\beta(\gamma(a)), \beta_{scope}, true)\} \cup \\ &\{(\beta(\gamma(ch)), \beta_{scope}, true)\} \cup \\ &\{(\beta(\gamma(eh)), \beta_{scope}, true)\} \cup \\ &\{(\beta(\gamma(fh)), \beta_{scope}, true)\} \cup \\ &\{(\beta(\gamma(th)), \beta_{scope}, true)\} \cup\end{aligned} \tag{6.5.2}$$

The edges created for the <scope> activity are the union of $\theta(scope)$ with the edges created by mapping the enclosed primary activity and by mapping the handlers.

$$E_{scope} = \theta(scope) \cup \theta(a) \cup \theta(ch) \cup \theta(eh) \cup \theta(fh) \cup \theta(th) \tag{6.5.3}$$

6.5.1 Mapping Compensation Handler

A <compensationHandler> is a wrapper for an activity that performs the compensation [16]. We denote a compensation handler as:

$$A_{ch} = \{a\}$$

Thus, the set of nodes V_{ch} resulted by mapping the compensation handler is defined as follows:

$$V_{ch} = \gamma(ch) = \{\alpha_{ch}, \beta_{ch}\} \cup \gamma(a) \tag{6.5.4}$$

The mapping function $\theta(ch)$ is defined as follows:

$$\begin{aligned}\theta(ch) = &\{(\alpha_{ch}, \alpha(\gamma(a)), true)\} \cup \\ &\{(\beta(\gamma(a)), \beta_{ch}, true)\}\end{aligned} \tag{6.5.5}$$

The edges created for the `<compensationHandler>` are the union of $\theta(ch)$ with the edges created by mapping the enclosed activity.

$$E_{ch} = \theta(ch) \cup \theta(a) \tag{6.5.6}$$

6.5.2 Mapping Event Handlers

Similarly to the mapping of a `<pick>` activity, `<eventHandlers>` can be mapped as follows:

$$\begin{aligned}A_{eh} =& A_{event} \cup A_{alarm} \\ =& \{(o_{event_1}, a_1), (o_{event_2}, a_2), ..., (o_{event_m}, a_m)\} \cup \\ & \{(o_{alarm_1}, a_{m+1}), (o_{alarm_2}, a_{m+2}), ..., (o_{alarm_n}, a_{m+n})\} \\ & 1 \leq m \text{ and } 1 \leq n\end{aligned}$$

Thus, the set of nodes V_{eh} resulted by mapping the event handler is defined as follows:

$$\begin{aligned}V_{eh} = \gamma(eh) = &\{\alpha_{eh}, \beta_{eh}\} \cup \bigcup_{i=1}^{m} \left\{\alpha_{o_{event_i}}, \beta_{o_{event_i}}\right\} \cup \\ &\bigcup_{j=1}^{n} \left\{\alpha_{o_{alarm_j}}, \beta_{o_{alarm_j}}\right\} \cup \bigcup_{k=1}^{m+n} \gamma(a_k)\end{aligned} \tag{6.5.7}$$

The mapping function $\theta(eh)$ is defined as follows:

$$\begin{aligned}\theta(eh) = &\bigcup_{i=1}^{m}\{(\alpha_{eh},\alpha(o_{event_i}),true)\}\cup\\ &\bigcup_{j=1}^{n}\left\{(\alpha_{eh},\alpha(o_{alarm_j}),true)\right\}\cup\\ &\bigcup_{i=1}^{m}\{(\alpha(o_{event_i}),\alpha(\gamma(a_i)),true)\}\cup\\ &\bigcup_{j=1}^{n}\left\{(\alpha(o_{alarm_j}),\alpha(\gamma(a_{m+j})),true)\right\}\cup\\ &\bigcup_{i=1}^{m}\{(\beta(\gamma(a_i)),\beta(o_{event_i}),true)\}\cup\\ &\bigcup_{j=1}^{n}\left\{(\beta(\gamma(a_{m+j})),\beta(o_{alarm_j}),true)\right\}\cup\\ &\bigcup_{i=1}^{m}\{(\beta(o_{event_i}),\beta_{eh},true)\}\cup\\ &\bigcup_{j=1}^{n}\left\{(\beta(o_{alarm_j}),\beta_{eh},true)\right\}\end{aligned} \tag{6.5.8}$$

The edges created for the <eventHandlers> activity are the union of $\theta(eh)$ with the edges created by mapping the enclosed activities in each branch.

$$E_{eh} = \theta(eh)\cup\bigcup_{k=1}^{m+n}\theta(a) \tag{6.5.9}$$

6.5.3 Mapping Fault Handlers

A <faultHandlers> is a wrapper for a set of specific fault handling activities defined by <catch> constructs or a default fault handling logic defined by the <catchAll> construct. We denote the fault handlers as a tuple of specific handlers $(fault, activity)$ and the default handler $(undefined, a)$. As the default fault handler catches all the faults that have not been caught by a more specific fault handler, thus, the element in the tuple is set to $undefined$. In the

following we use (f_i to denote fault handlers and a_i to denote the corresponding activity for fault handling.

$$A_{fh} = \{(f_1, a_1), (f_2, a_2), ..., (f_k, a_k), (undefined, a_{k+1})\}$$

Thus, the set of nodes V_{fh} resulted by mapping the fault handlers is defined as follows:

$$V_{fh} = \gamma(fh) = \{\alpha_{fh}, \beta_{fh}\} \cup \bigcup_{i=1}^{k+1} \gamma(a_i) \tag{6.5.10}$$

The mapping function $\theta(fh)$ is defined as follows:

$$\begin{aligned} \theta(fh) = & \bigcup_{i=1}^{k} \{(\alpha_{fh}, \alpha(\gamma(a_i)), f_i)\} \cup \\ & \{(\alpha_{fh}, \alpha(\gamma(a_{k+1})), undefined)\} \\ & \bigcup_{i=1}^{k+1} \{(\beta(\gamma(a_i)), \beta_{fh}, true)\} \end{aligned} \tag{6.5.11}$$

The edges created for the `<faultHandlers>` are the union of $\theta(fh)$ with the edges created by mapping each enclosed activity in each fault handler.

$$E_{fh} = \theta(fh) \cup \bigcup_{i=1}^{k+1} \theta(a_i) \tag{6.5.12}$$

6.5.4 Mapping Termination Handler

A `<terminationHandler>` can be denoted as:

$$A_{th} = \{a\}$$

Thus, the set of nodes V_{th} resulted by mapping the termination handler is defined as follows:

$$V_{th} = \gamma(th) = \{\alpha_{th}, \beta_{th}\} \cup \gamma(a) \tag{6.5.13}$$

The mapping function $\theta(th)$ is defined as follows:

$$\begin{aligned} \theta(th) = &\{(\alpha_{th}, \alpha(\gamma(a)), true)\} \cup \\ &\{(\beta(\gamma(a)), \beta_{th}, true)\} \end{aligned} \tag{6.5.14}$$

The edges created for the <terminaationHandler> are the union of $\theta(th)$ with the edges created by mapping its enclosed activity.

$$E_{th} = \theta(th) \cup \theta(a) \tag{6.5.15}$$

6.6 Mapping the Root Elements

Recall that in our mapping framework we consider only activities, activity containers, and their static structure. When only considering these constructs for the mapping, we can reuse the rules of mapping a <scope> activity to map the <process> element.

Different from the <process> element, the <fragment> element allows to immediately enclose more than one activities, <link>s, compensation and termination handlers. Thus, we need to introduce new mapping rules for the <fragment> element. We consider a fragment as a tuple $(A_{fragment}, L_{fragment}, A_{handlers})$. Note that $L_{fragment}$ denotes <link>s that are immediately used within the <fragment> element.

$$\begin{aligned} A_{fragment} &= \{a_1, a_2, ..., a_n\} \\ L_{fragment} &= \{l_1, l_2, ..., l_m\} \\ A_{handlers} &= \{(ch, eh, fh, th)\} \end{aligned}$$

Thus, the set of nodes $V_{fragment}$ resulted by mapping the `<fragment>` activity is defined as follows:

$$\begin{aligned} V_{fragment} =& \gamma(fragment) \cup \gamma(handlers) \\ =& \{\alpha_{fragment}, \beta_{fragment}\} \cup \bigcup_{i=1}^{n} \gamma(a_i) \cup \\ & \gamma(ch) \cup \gamma(eh) \cup \gamma(fh) \cup \gamma(th) \end{aligned} \tag{6.6.1}$$

Similar to mapping a `<flow>` activity, we need also to identify the start and end activities that are immediately enclosed in the `<fragment>` element. They are defined as follows:

$$\begin{aligned} A_{start} &= \{a_i | L_{a_i}^{incoming} = \emptyset\} \\ A_{end} &= \{a_i | L_{a_i}^{outgoing} = \emptyset\} \setminus A_{start} \end{aligned}$$

The mapping function $\theta(fragment)$ is defined as follows:

$$\begin{aligned} \theta(fragment) = & \bigcup_{a_i \in A_{start}} \{(\alpha_{fragment}, \alpha(\gamma(a_i)), true)\} \cup \\ & \{(\alpha_{fragment}, \alpha(\gamma(ch))), undefined\} \cup \\ & \{(\alpha_{fragment}, \alpha(\gamma(eh))), true\} \cup \\ & \{(\alpha_{fragment}, \alpha(\gamma(fh))), true\} \cup \\ & \{(\alpha_{fragment}, \alpha(\gamma(th))), true\} \cup \\ & \bigcup_{j=1}^{m} \{(\beta(\gamma(\pi_1(l_j)))), \alpha(\gamma(\pi_2(l_j))), \varphi(l_j)\} \cup \\ & \{(\beta(\gamma(ch)), \beta_{fragment}, true)\} \cup \\ & \{(\beta(\gamma(eh)), \beta_{fragment}, true)\} \cup \\ & \{(\beta(\gamma(fh)), \beta_{fragment}, true)\} \cup \\ & \{(\beta(\gamma(th)), \beta_{fragment}, true)\} \cup \\ & \bigcup_{a_k \in A_{end}} \{(\beta(\gamma(a_k)), \alpha_{fragment}, true)\} \end{aligned} \tag{6.6.2}$$

The edges created by the mapping are the union of $\theta(fragment)$ with the edges created by mapping each immediately enclosed activity and the handlers within the `<fragment>` element.

$$E_{fragment} = \theta(fragment) \cup \bigcup_{i=1}^{n} \theta(a_i) \cup \theta(ch) \cup \theta(eh) \cup \theta(fh) \cup \theta(th) \tag{6.6.3}$$

Chapter 7
Querying BPEL Fragments

This chapter presents a novel mechanism for querying BPEL fragments and process models. In Section 7.2 we analyze different matching types of structural matchmaking. In Section 7.3 we introduce the preliminaries for the query algorithm. In Section 7.4 we present the query algorithm for approximate structural matchmaking of BPEL fragments and process models.

7.1 Introduction

In this chapter we focus on querying structural information of process fragments or models to support reuse in process modeling. Based on an empirical study conducted in the e-science community, users tend to use structural information to discover process models [63]. However, querying structural information of processes models could also be of interest in other application areas. The structure of a process model refers to the way in which process activities are connected together, arranged or organized. Process structure does not consider the business semantics of the process model, such as transition conditions, join conditions, etc. It is a snapshot of the static construction of the process model.

One interesting usage scenario for structural query is auto-completion. Based on lessons learned from the reuse practices, the main barrier to a successful implementation of reuse is not technological but cultural [39], i.e. users of

reusable artifacts are not trained to apply the *design by reuse* principle [114]. One of the major issues is that users do not know where to find reusable artifacts. Even with a centralized reuse repository in place, users may not be willing to make the effort to search for reusable artifacts, but tend to build a new one on their own. Motivated by this knowledge, auto-completion aims to promote the reuse of proven process models and fragments stored in a process repository.

Assume that a user begins to model a process model from scratch. While the user is drawing modeling elements on the canvas, in background, the process modeling tool could capture the already modeled part and send it as a query request to the process repository. The process repository would try to find process models or fragments that contain the same or similar structure as the query request. In case the query was successful, the process repository could return the results sorted by the similarity degree of their process structure. Otherwise, there exist no process models or fragments that can be reused.

To provide a better support for reuse, the retrieve mechanism in the process repository should not only be able to return process models and fragments that match exactly the process structure of the query request, but also be able to return process models and fragments that approximately match the process structure of the query request (see. Section 7.2.2).

In the following sections we introduce an algorithm for querying BPEL process models and fragments that approximately match the process structure of the query request. In our algorithm we do not consider incomplete links for query processing. We leave it as future work.

7.2 Types of Structural Matchmaking

In a graph-based approach for structural matchmaking, we consider the following matching types: (sub)graph isomorphism, inexact match, and vertex coverage.

7.2.1 Isomorphism

Isomorphism represents the case where the process graph matches exactly to the query request. In a graph representation the graph isomorphism can be defined as follows.

Let $Q = (V_Q, E_Q)$ be the query graph, and $P = (V_P, E_P)$ be the process graph. The process graph P is said to be a isomorphic matchmaking of the query graph Q, if there exist a bijective function $f : V_Q \to V_P$ such that $(f(u), f(v)) \in E_P$ if and only if $(u, v) \in E_Q$. If there is a graph isomorphism between Q and P then P is said to be isomorphic to Q, written $P \cong Q$.

7.2.2 Approximate Match

To provide a better support for reuse, the retrieve mechanism in the process repository should not only be able to return process models and fragments that are isomorphic to the request of a user, but also be able to deliver similar ones. In case no isomorphic process models or fragments can be found, query constraints should be relaxed and results with slight deviations should be returned to users.

Approximate match requires that i) for a subset of the nodes in the query graph, each of them has a distinctive matching node in the graph of a process model or fragment; ii) for each pair of the nodes in the subset of the query graph, their respective matching nodes in the process graph have the same parent-child or ancestor-descendant relationship as the nodes in the query graph.

Assume that $Q = (V_Q, E_Q)$ is a query graph and $P = (V_P, E_P)$ is a process graph, P is said to be an approximate match of Q if and only if the following conditions satisfy:

- for a subset $\hat{V}_Q \subseteq V_Q$, there exists injective function $f : \hat{V}_Q \to V_P$
- $\forall u$ and $v \in \hat{V}_Q$, if there exists a directed path $(u, v_1, v_2, ..., v_k, v)$ where $v_1, v_2, ..., v_k \in V_Q$, then there exists a directed path $(f(u), v_{k+1}, v_{k+2}, ..., v_{k+m}, f(v))$ in P, where $f(u), v_{k+1}, v_{k+2}, ..., v_{k+m}, f(v) \in V_P$

7.2.3 Vertex Coverage

When further relaxing the matchmaking criteria of the edges, we have the matching type *vertex coverage*. Vertex coverage requires that the process graph contains all the nodes in the query graph that have the same values of the matching criteria, e.g. the same activity name, the same activity type, etc. As isomorphic graphs and exact matches contain all the nodes in the query graph, they are also special cases of vertex coverage.

Let $Q = (V_Q, E_Q)$ be the query graph, and $P = (V_P, E_P)$ be the process graph. The process graph P is said to be a vertex coverage on Q if there exists an injective function $f : V_Q \rightarrow V_P$.

7.3 Preliminaries

7.3.1 Matching Semantics

Let $Q = (V_Q, E_Q)$ be a query graph and $P = (V_P, E_P)$ be a process graph. We say an *assignment* over V_Q is a function $\mu : V_Q \rightarrow V_P \cup \{\perp\}$. We use the symbol $\perp$ to indicate an undefined node. If $\mu(v) \neq \perp$, we say that the query node v has a matching node in P or the query node v is bound. Otherwise, the query node v has no matching nodes in P and its value is undefined.

An edge is a *matching edge* , if and only if $\mu(u) \neq \perp$, $\mu(v) \neq \perp$, $(u,v) \in E_Q$, i.e. $\mu(u)$ is ancestor of $\mu(v)$ where $\mu(u), \mu(v) \in V_P$.

Figure 7.1 shows an example of a query graph Q and a process graph P. The labels in each graph represent the identifiers of the nodes. The same letter with different subscripts indicates different instances of the matching nodes. For example, the nodes o_1, o_2 in the process graph P are two different matching instances of the node o in the query graph Q. The matchmaking can be based on different matching predicate like activity names, activity types, etc.

In this thesis, we consider the level of a node q in a rooted, directed, acyclic graph (DAG) as the length of the longest path from the root to q. We use q_r to denote the root of Q and call it the $level - 0$ node. Similarly, we call a node q_i

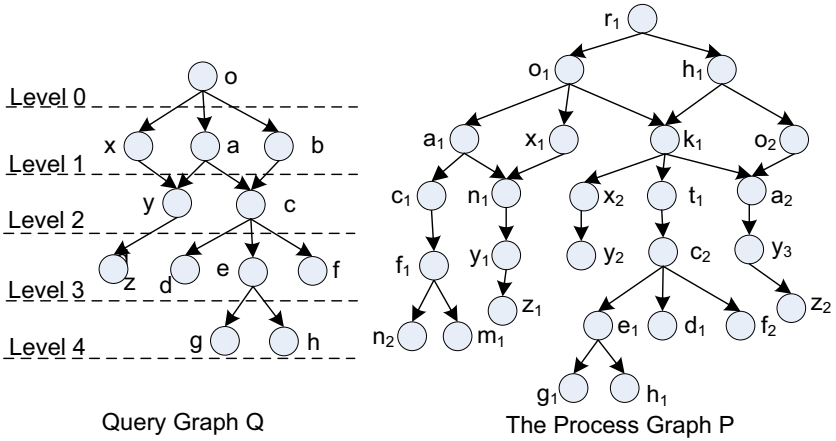


Fig. 7.1 The query graph Q and the process graph P.

at level i a $level-i$ node. We use q_{ij} to denote both the level and the instance index, where i indicates the level of the node and j indicates the instance index of the node in the graph.

If there exist an assignment that maps a subgraph rooted at q_{ij} to a subgraph of P rooted at p_k, then we call this assignment a $level-i$ assignment . The assignment of the root element of the subgraph q_{ij} $\mu : q_{ij} \rightarrow p_k$ is called the start assignment of the $level-i$ assignment . We use the representation $[q_{ij}/p_k]$ to denote the start assignment.

For example, for the two graphs shown in Figure 7.1, a start assignment for the graph Q is $[o/o_1]$. And the start assignment for the subgraph rooted at b is $[b/\perp]$, where the symbol $\perp$ indicates that b has no matching node in P.

An assignment that maps the whole query graph Q to the process graph P is called a global assignment. As a $level-0$ assignment maps the whole query graph to the process graph, it is a global assignment. An assignment that maps only a subgraph of Q to a subgraph of P a local assignment. A $level-i$ assignment ($i > 0$) maps only a subgraph rooted at a node of level i of the query graph to a subgraph of the process graph, thus, it is a local assignment.

Let's consider again the example shown in Figure 7.1. For the query graph Q there are 8 $level-0$ assignments (global assignments). We call them $(level-0)_1$,

Level	o	x	y	z	a	b	c	d	e	f	g	h
$(\text{Level-0})_1$	o_1	x_1	y_1	z_1	a_1	$\perp$	c_1	$\perp$	$\perp$	f_1	$\perp$	$\perp$
$(\text{Level-0})_2$	o_1	x_1	y_1	z_1	a_2	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$(\text{Level-0})_3$	o_1	x_1	y_3	z_2	a_2	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$(\text{Level-0})_4$	o_1	x_2	y_1	z_1	a_1	$\perp$	c_1	$\perp$	$\perp$	f_1	$\perp$	$\perp$
$(\text{Level-0})_5$	o_1	x_2	y_2	$\perp$	a_1	$\perp$	c_1	$\perp$	$\perp$	f_1	$\perp$	$\perp$
$(\text{Level-0})_6$	o_1	x_2	y_2	$\perp$	a_2	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$(\text{Level-0})_7$	o_1	x_2	y_3	z_2	a_2	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$(\text{Level-0})_8$	o_2	$\perp$	y_3	z_2	a_2	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
...												
$(\text{Level-2})_1$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	c_2	d_1	e_1	f_2	g_1	h_1
$(\text{Level-2})_2$	$\perp$	$\perp$	y	z	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
...												

Fig. 7.2 Some example assignments found between the query graph Q and the process graph P.

$(level-0)_2$, ..., $(level-0)_8$ assignments as shown in Figure 7.2. Also, we have two $level-2$ assignments in the table: $(level-2)_1$ is a local assignment of the subgraph rooted at c; $(level-2)_2$ is a local assignment of the subgraph rooted at y.

7.3.2 Similarity Measurement

Based on the discussion on matching types in the previous section we introduce in this section three approaches for similarity measurement of approximate matches: vertex similarity, structural similarity, and hybrid similarity. The similarity measurements will be used for the query algorithm that we will discuss in the following sections.

Structural similarity

Structural similarity computes the similarity of two graphs based on the number of their matching edges, because a matching edge indicates both the match-

ing nodes and their ancestor-descendant relationship. For a subgraph G' of the query graph G, if all the edges in G' are matching edges, then we call the subgraph G' a *similarity component*. If there exists no ambiguity, we use *component* for short in the following discussions.

When computing the similarity between two (sub)graphs, we use a set of disjoint components. Two components $C_i = (V_{C_i}, E_{C_i})$ and $C_j = (V_{C_j}, E_{C_j})$ are disjoint, if they do not have common edges, i.e. $E_{C_i} \cap E_{C_j} = \emptyset$. Using disjoint components ensures that the same matching edges are not counted twice during similarity measurement.

Let $\{C_{m1}, C_{m2}, ..., C_{mn}\}$ be a set of disjoint components of the query graph Q. Assume that a set of assignments $U_m = \{\mu_{mj} | \mu_{mj} : C_{mj} \to P_{mj}\}$ maps each component to a subgraph of a process graph P, where P_{mj} is a subgraph of P, $m, n \in \mathbb{N}$, and $1 \leq j \leq n$. We use $|\mu_{mj}|$ to denote the number of matching edges of the assignment μ_{mj} and call it the *matching size* of μ_{mj}. We use $|E_Q|$ to denote the number of edges in the graph Q. The structural similarity of the process graph P and the query graph Q can be computed as follows:

$$S_{structural} = \frac{|\mu_{mk}| + k_{structural} * \sum(|\mu_{mi}|)}{|E_Q|} \tag{7.3.1}$$

with $\mu_{mi} \in U_m \backslash \{\mu_{mk}\}$, where μ_{mk} is the biggest matching component with $\mu_{mk} \in \{\mu_{ml} | |\mu_{ml}| = max(|\mu_{m1}|, |\mu_{m2}|, ..., |\mu_{mn}|)\}$ and $k_{structural}$ is a customizable weighting factor with $0 < k_{structural} < 1$, which should be determined by users.

To explain the reason of introducing the weighting factor $k_{structural}$, let's consider the following example. Figure 7.3 illustrates a process graph P' and a query graph Q'. In the example, we find three assignments as shown in Figure 7.4. Each of these assignments maps a component of Q' to a subgraph of P'. For the three assignments we can identify three similarity components that have no common edges (as shown in Figure 7.3). Thus, they are disjoint components of Q'.

If we simply sum up the number of matching edges of the components and divide it by the size of query graph Q', i.e. $(|\mu_{C_1}| + |\mu_{C_2}| + |\mu_{C_3}|)/|E_{Q'}| = (3 +$

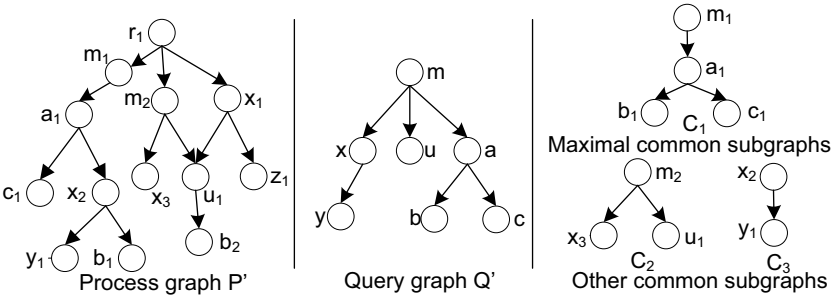


Fig. 7.3 Without the weighting factor the structural similarity maybe distorted.

Level	m	x	u	a	y	b	c
$(\text{Level-0})_1$	m_1	$\perp$	$\perp$	a_1	$\perp$	b_1	c_1
$(\text{Level-0})_2$	m_2	x_3	u_1	$\perp$	$\perp$	$\perp$	$\perp$
$(\text{Level-1})_1$	$\perp$	x_1	$\perp$	$\perp$	y_1	$\perp$	$\perp$

Fig. 7.4 Three assignments found between the process graph P' and the query graph Q'.

$2+1)/6$, then we get a similarity equals to 1. This result would indicate that the structure of the query graph Q' was identical with the structure of the process graph P', which is obviously not true. Therefore, we introduce the weighting factor $k_{structural}$ to prevent this kind of distorted similarity. The value of the weighted factor ranges from 0 to 1 and should be determined by users based on experimentations.

However, we should not apply the weighting factor $k_{structural}$ to all the components that are used to calculate the similarity. For example, assume that a process graph P' is isomorphic to the query graph Q'. Thus, the only matching component is Q' itself, which is then also the biggest matching component. If we apply the weighting factor $k_{structural}$ to the biggest matching component Q', then similarity of P'' and Q' would be $k_{structural} * |E_{Q'}|/|E_{Q'}| = k_{structural}$ with $0 < k_{structural} < 1$. In other words, the value of the structural similarity of P'' and Q' is smaller than 1. Apparently, this is not true. Because P'' and Q' are isomorphic, the actual value of their structural similarity should be 1. For that reason, we use a maximal component (i.e. a component with the most num-

ber of matching edges) without the weighting factor and apply $k_{structural}$ to the other disjoint components.

Note that if the similarity between a query graph and a process graph equals to 1, it does not necessarily indicate that the query graph and the process graph are isomorphic. The process graph could also be an exact match of the query graph.

Back to the example shown in Figure 7.3, let the value of the weighting factor $k_{structural}$ be 0.9. The structural similarity between P' and Q' can be calculated as $(3+0.9*(2+1))/6 = 0.95$.

Vertex Similarity

Vertex similarity computes the similarity between a query graph and a process graph solely based on the number of their matching nodes. This approach ignores the ancestor-descendant relationships of the matching nodes. Vertex similarity is especially useful, when a process modeler wants to find process models or fragments that contain a subset of process activities as specified in the query request but does not care how they are orchestrated.

Vertex similarity can be calculated as follows:

$$S_{vertex} = \frac{|V_Q^{match}|}{|V_Q|} \tag{7.3.2}$$

$|V_Q^{match}|$ denotes the number of nodes in the query graph Q that have a matching node in the process graph P and $|V_Q|$ denotes the number of nodes of the graph Q.

For the example shown in Figure 7.3, the vertex similarity can be calculated as $S_{vertex} = 7/7 = 1$, which means all the nodes in the query graph Q' have matching nodes in the process graph P'.

Hybrid similarity

Hybrid similarity takes both structural and vertex similarities into consideration.

The hybrid similarity is defined as follows:

$$S_{hybrid} = \frac{k_1 * (|\mu_{mk}| + k_{structural} * \sum(|\mu_{mi}|)) + k_2 * |V_Q^{match}|}{k_1 * |E_Q| + k_2 * |V_Q|} \quad (7.3.3)$$
$$\text{where } 0 \leq k_1, k_2 \leq 1 \text{ and } k_1 + k_2 = 1$$

In the formula above we introduced two weighting factors: (i) k_1 is the weighting factor for structural similarity; (ii) and k_2 is the weighting factor for vertex similarity. These two weighting factors allow users to define which proportion the structural and vertex similarities should be considered in the overall similarity measurement. Which of the weighting factors should be larger depends on whether the structural similarity or the vertex similarity is more important for users. Both structural and vertex similarity are special cases of hybrid similarity. We will use the numerator in the Formula (7.3.3) as the *match size* later in this chapter.

Let's consider the example shown in Figure 7.3 again. Let the weighting factor for structural similarity $k_1 = 0.7$ and the weighting factor for vertex similarity $k_1 = 0.3$. Also, let $k_{structural} = 0.9$. Then we can calculate the hybrid similarity $S_{hybrid} = (0.7 * (3 + 0.9 * (2 + 1)) + 0.3 * 7) / (0.7 * 6 + 0.3 * 7) \approx 0.97$. Apparently, the vertex similarity increases the overall similarity than only considering the structural similarity.

7.3.3 Data Structure

Our query algorithm works on a novel data structure called *solution stream*, which captures all the necessary data needed in different phases of the query algorithm. In the following we use the example shown in Figure 7.1 to explain the data structure.

Each node in the query graph has a solution stream. Figure 7.5 illustrates a snippet of the solution streams initialized by using the example shown in Figure 7.1. Let's begin with the root o of the query graph Q. The solution stream S_o for the root o comprises of: (i) a list of stream items; (ii) the maximal match size of its stream items; (iii) a set of assignments (*maxAssignmentSet*) of the subgraph rooted at o that have the maximal matching size.

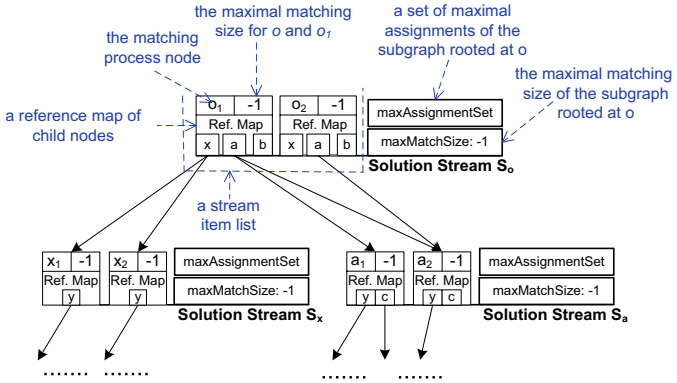


Fig. 7.5 A part of initialized solution streams.

Each matching node of o in the process graph P has a stream item. In the example shown in Figure 7.1 the root o has two matching nodes in P, i.e. o_1 and o_2. Thus, the stream item list consists of two stream items.

A stream item stores the information of the matchmaking of a node in the query graph and the respective node in the process graph. Taking the root o and a matching node o_1 as an example the stream item $si_o^{o_1}$ contains the following data:

- the matching node o_1 in the process graph P;
- the maximal match size of the assignments that maps the subgraph rooted at o to the subgraph rooted at o_1 with an initial value of -1;
- a reference map of child nodes of o in the query graph, i.e. x, a, and b. As (o_1, x_1) is a matching edge, the reference map stores also a reference from the child node x to the stream item $si_x^{x_1}$. Similarly, the reference maps stores

also the reference from x to the stream item $si_x^{x_1}$, from a to the stream items $si_a^{a_1}$ and $si_a^{a_2}$.

Likewise, further solution streams and stream items can be created as discussed above. As solution stream may contain more than one stream items, we use *maxMatchSize* to store the maximal matching size in the initialization phase. It allows the algorithm to retrieve the maximal matching size without having to go through each assignment and compute the maximal matching size again. The set *maxAssignmentSet* stores the corresponding assignments with the maximal matching size.

7.4 An Approximate Query Algorithm

In this section we present an approximate query algorithm. The basic idea of our algorithm is to find all the maximal local assignments for subgraphs rooted at each query node and then to examine which combinations of the maximal local assignments results in an overall maximal similarity value.

For example, as shown in Figure 7.2 the maximal $level-0$ assignments do not reflect the total similarity between P and Q. The two graphs P and Q have an isomorphic common subgraph rooted at c. But this local assignment is not included in any of the $level-0$ assignments.

If we combine the part of the assignment $(level-0)_1$ (assignment for o, x, y, z, a, and b) with the part of the assignment $(level-2)_1$ (assignment for c, d, e, f, g and h), then the new global assignment would result in a larger similarity value than each individual assignment would. Thus, we cannot simply compute the similarity between the query graph and the process graph from maximal $level-0$ assignments. In the following, We introduce a corresponding approach for measuring the similarity. The algorithm comprises of three phases: the initialization phase, the assignment phase, and the combination phase.

7.4.1 The Initialization Phase

The initialization phase creates for each query node a solution stream and populate the solution streams with the data needed for the following two phases. The procedure of the initialization phase is shown in Algorithm 22. At the beginning of the algorithm (Algorithm 22, line 2) we use a sorted array ($queryNodes[]$) of all the query nodes in the query graph Q. The query nodes in $queryNodes[]$ are sorted in an descending order of their topological levels in Q. The rest of the algorithm can be divided into two logical parts.

The first logical part (Algorithm 22, line 5-24) traverses the query graph Q. For each query node q_i the algorithm creates a solution stream S_{q_i} (Algorithm 22, line 6). If the query node q_i is a leaf node, then the subgraph rooted at q_i does not contain any edges. Thus, its structural matching size is 0. Otherwise, its structural matching size is undefined. In this case, we set the value of its maximal structural similarity to -1.

Then the algorithm compares the query node q_i with each node p in the process graph P (Algorithm 22, line 11). If the predicates of q_i (e.g. activity name, variable name, partner link type, etc.) being compared match those of the node p, then the algorithm creates a stream item $si_{q_i}^{p}$ (Algorithm 22, line 14) and stores the matching process node p in it (22, line 15). Analogously, if the query node q_i is a leaf node, then we set its maximal match size to 0; otherwise, we set it to -1 (Algorithm 22, line 16-19). Recall that a query node may have more than one matching process node. The boolean variable $vertexMatchFlag$ is used to ensure that a matching query node will be counted only once. If the query node q_i does not match the process node p, then the algorithm does not create the stream item for q_i.

The second part (Algorithm 22, line 25-30) of the algorithm examines whether the matching nodes in the process graph have the same ancestor-descendant relationships as their corresponding query nodes.

Assume that q_i is a query node and q_i^{child} is a child node of q_i. Let p_i in the process graph P be a matching node of q_i and p_i' in the process graph P be a matching node of q_i^{child}. If q_i is an ancestor of q_i^{child} in the process graph P, then (q_i, q_i^{child}) is a matching edge. Thus, the stream item $si_{q_i}^{p_i}$ stores a reference

Algorithm 22: Initialization phase

Input: a query graph Q and a process graph P
Output: Initialized solution streams for the query graph Q

```
1  begin
2      queryNodes[] = [q_0, q_1, q_2, ..., q_k];
3      vertexCounter = 0;
4      vertexMatchFlag = false;
5      for i = 0 to k do
6          create a new solution stream S_{q_i};
7          if isLeaf(q_i) then
8              S_{q_i}.maxMatchSize = 0;
9          else
10             S_{q_i}.maxMatchSize = -1;
11         for each p in V_P do
12             if isMatching(q_i, p) then
13                 vertexMatchFlag = true;
14                 create a new stream item si^p_{q_i};
15                 si^p_{q_i}.processNode = p;
16                 if isLeaf(q_i) then
17                     si^p_{q_i}.maxMatchSize = 0;
18                 else
19                     si^p_{q_i}.maxMatchSize = -1;
20                 add all the child nodes of q_i in si^p_{q_i};
21             add si^p_{q_i} to S_{q_i};
22         if vertexMatchFlag = true then
23             vertexCounter = vertexCounter + 1;
24             vertextMatchFlag = false;
25         for each solution item si^p_{q_i} in the solution stream S_{q_i} do
26             p = si_{q_i}.prcessNode;
27             for each child node q_i^child of q_i in si_{q_i} do
28                 p' = si_{q_i^child}.processNode;
29                 if isAncestor(p, p') then
30                     add reference from q_i^child to si^{p'}_{q_i^child} in the reference
                       map of si^p_{q_i};
31 end
```

from the child node q_i^{child} to the stream item $si^{p'_i}_{q_i^{child}}$. The reference indicates that the edge (q_i, q_i^{child}) is a matching edge.

The worst case time complexity of the initialization phase is the sum of the time complexity of the two parts in the algorithm, i.e. $O\left(|V_Q||V_P| + |V_Q|cb^2\right)$, where $|V_Q|$ and $|V_P|$ denote the order of the query graph and the process graph respectively, c is the average number of the child nodes of the query nodes, and b is the average number of solution items of the query nodes.

7.4.2 The Assignment Phase

The assignment phase computes the maximal assignments for all the subgraphs that are rooted at a query node in the query graph Q (maximal level-i assignments). A maximal level-i assignment of a query node q_i can be efficiently calculated by combining the maximal assignments of its child nodes. Thus, we follow a bottom-up approach and begin with the leaf nodes in the query graph and work towards the root.

Algorithm 23: Assignment phase

Input: a query graph Q and a process graph P
Output: the maximal assignments for each query node in Q

```
1 begin
2     queryNodes[] = [q_0, q_1, q_2, ..., q_k];
3     sort the array queryNodes[];
4     for i = 0 to k do
5         if isLeaf(q_i) then
6             compute maximal assignment for leaf nodes;
7         else
8             compute maximal assignments for non-leaf nodes;
9 end
```

As shown in Algorithm 23, the algorithm takes a query graph Q and a process graph P as input and calculates the maximal assignments for each query

node in Q. At the beginning of the algorithm we sort the query nodes in a descending order of their levels and store them in an array (Algorithm 23, line 3). Then we iterate each element in the array. If the query node being processing is a leaf node, then we invoke the Algorithm 24; otherwise, we invoke the Algorithm 25 for further processing.

If a query node q_i is a leaf node in the query graph, then we first examine whether its solution stream has stream items ((Algorithm 24, line 2). If a query node has stream items, then the query node has matching node in the process graph. Otherwise, the query node has no matching node in the process graph.

If q_i has stream items, then the algorithm iterates the stream items and create for each stream item an assignment from q_i to its corresponding matching process node p_i (Algorithm 24, line 2-5). Otherwise, it means that q_i has no matching process nodes and the algorithm assigns p_i to $\perp$ (Algorithm 24, line 6-7). In both cases, there are no matching edges. Therefore, the maximal matching size of each solution item si_{q_i} is 0 (Algorithm 24, line 8) and μ_{q_i} is the maximal assignment of q_i (Algorithm 24, line 9-10).

Algorithm 24: Compute maximal assignments for leaf nodes

```
1  begin
2      if S_{q_i} has stream items then
3          for each stream item si_{q_i} do
4              p_i = si_{q_i}.processNode;
5              μ_{q_i} = [q_i/p_i];
6      else
7          μ_{q_i} = [q_i/⊥];
8      si_{q_i}.maxMatchSize = 0;
9      tempMaxAssignmentSet[q_i/p_i] = {μ_{q_i}};
10     maxAssignmentSet[q_i] = maxAssignmentSet[q_i] ∪ {μ_{q_i}}
11 end
```

If a query node q_i is an inner node, we can calculate the maximal assignments for it by combining the maximal assignments of its child nodes (Algorithm 25). The start assignment of the maximal assignment μ_{p_i} is $[q_i/p_i]$ (Al-

gorithm 25, line 5), which is also the current temporary maximal assignment of q_i (Algorithm 25, line 6).

Algorithm 25: Compute maximal assignments for inner nodes

```
1  begin
2      if S_{q_i} has stream items then
3          for each stream item si_{q_i} do
4              p_i = si_{q_i}.processNode;
5              μ_{q_i} = [q_i/p_i];
6              tempMaxAssignmentSet[q_i, p_i] = {μ_{q_i}};
7              processedDescendants = {};
8              let childNodes be a sorted list of child nodes of q_i;
9              for each child node q_c in childNodes do
10                 let descendants_{q_c} be the set of all descendants of q_c;
11                 commonNodes =
                   (processedDescendants ∩ descendants_{q_c});
12                 if si_{q_i} has no references to any stream items of q_c then
13                     Set assignments to ⊥;
14                 else
15                     Compute the combination of assignments;
16                 processedDescendants.add(descendants_{q_c});
17             Set the value of maxAssignmentSet;
18 end
```

Subgraphs rooted at child nodes may overlap with each other. If we simply combine the assignments of the subgraphs, a larger assignment may be overwritten by a smaller one, which would lower the actual similarity value between the query graph and the process graph. Figure 7.6 shows an exmaple with a query graph Q and a process graph P. The symbole $'$ in the labeling of a process node indicates that the process node has a matching node in the query graph Q. For exmaple, the process node a' is a matching node of the query node a in the query graph Q. Assume that we have already calculated the maximal assignments for subgraphs rooted at b with $\mu_b = \{[b/b'][d/d'][e/e']\}$ and the maximal assignments for subgraphs rooted at c with $\mu_c = \{[c/c'][d/\perp][e/\perp]\}$. Now let's calculate the maximal assignments of the subgraph rooted at a.

The start assignment of the subgraph rooted at a is $[a/a']$. To compute the maximal assignment of a we first combine its start assignment with μ_b. As a result we get $\mu_1 = \{[a/a'][b/b'][d/d'][e/e']\}$. The subgraph rooted at b has common nodes with the subgraph rooted at c, i.e. d and e. Simply combining μ_1 with μ_c would overwrite the assignments $[d/d'][e/e']$ by the assignments $[d/\perp][e/\perp]$. The combined assignment is

$$\mu_2 = \left\{ \left[a/a'\right] \left[b/b'\right] \left[c/c'\right] [d/\perp] [e/\perp] \right\}.$$

Due to the overwritting, we lost two matching edges in the assignment μ_2, i.e. (b,d) and (d,e). Therefore, when computing the maximal assignments for inner nodes, we have to make sure that none of the common nodes of the assignments with larger matching size are overwritten.

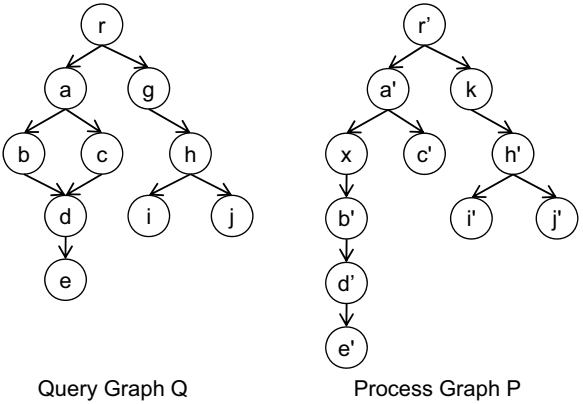


Fig. 7.6 Assignments of the subgraphs rooted at the child nodes may overlap with each other.

Thus, we first sort the child nodes of q_i based on the following conditions (Algorithm 25, line 8): (i) child nodes whose stream items are referenced by stream items of q_i are placed before child nodes whose stream items have no references; (ii) child nodes that are referenced by stream items of q_i are sorted in descending order of their maximal matching size. Condition (i) let us focus on processing matching process nodes that represent the same ancestor-descendant relationships as the query nodes. Condition (ii) ensures that assign-

ments with larger matching size are processed before the assignments with smaller matching size.

We then use the set *processedDescendants* to store all the descendants of q_i's child nodes that have been processed. Assume that the algorithm is going to process the next child node q_c of q_i. If the subgraph rooted at q_c has common nodes with already processed subgraphs, then the common nodes are the intersection of *processedDescendants* and the descendants of q_c (Algorithm 25, line 11).

When computing the combination of the child assignments, there exist two cases: (i) si_{q_i} has no references to si_{q_c}; (ii) si_{q_i} has references to si_{q_c}.

Case (i): if none of the stream items of q_c has been referenced by the stream item si_{q_i} (Algorithm 25, line 12), then the matching process node of q_i is not an ancestor of the matching process node of q_c. In other words, the edge (q_i, q_c) is not a matching edge. Thus, we invoke the Algorithm 26 to assign all the nodes of the subgraph rooted at q_c to $\perp$ (Algorithm 25, line 13).

Algorithm 26: Set assignments to $\perp$

```
1 begin
2     for each assignment μ_{q_i} in tempMaxAssignmentSet[q_i, p_i] do
3         if commonNodes is empty then
4             for each q_x ∈ V_{q_c} do
5                 μ_{q_i} = μ_{q_i} ⊕ μ_{q_x};
6         else
7             for each q_x ∈ (V_{q_c} \ commonNodes) do
8                 μ_{q_i} = μ_{q_i} ⊕ [q_x/⊥];
9 end
```

Algorithm 26 combines each assignments in $tempMaxAssignments[q_i, p_x]$ with assignments that maps each query nodes in the subgraph rooted at q_c to undefined ($\perp$). As discussed above, subgraphs rooted at the child nodes of q_i may overlap with each other. When combining their assignments, we have to examine whether they have common nodes.

We first introduce two operations that are needed in the following algorithms for combining two assignments: the *union* operation and the *difference* operation.

Definition 8. The Union Operation $\oplus$
Let $\mu : V_Q \rightarrow V_P \cup \perp$ and $\mu' : V'_Q \rightarrow V'_P \cup \perp$ be two assignments and $V_Q \cap V'_Q = \emptyset$. The union of the assignments μ and μ' is a function $\mu'' : V_Q \cup V'_Q \rightarrow V_P \cup V'_P \cup \perp$. We use the symbol $\oplus$ to denote the union operation. Thus, $\mu'' = \mu \oplus \mu'$. □

Definition 9. The Difference Operation $\ominus$
Let $\mu : V_Q \rightarrow V_P \cup \perp$ and $\mu' : V'_Q \rightarrow V'_P \cup \perp$ be two assignments. The difference of the assignments μ and μ' is a function

$$\mu'' = \begin{cases} V_Q \backslash V'_Q \rightarrow V_P \cup \perp & V'_Q \neq V_Q \\ undefined & V'_Q = V_Q \end{cases}$$

We use the symbol $\ominus$ to denote the difference operation. Thus, $\mu'' = \mu \ominus \mu'$. □

If *commonNodes* is empty, which means the subgraph being processed has no common nodes with already processed subgraphs (Algorithm 26, line 3), we can just combine the two assignments (Algorithm 26, line 5).

If *commonNodes* is not empty, then it means that some of the nodes in the subgraph rooted at q_c have already been assigned. Recall that the child nodes of q_i have been processed in a descending order of the maximal matching size of their assignments. Thus, the existing assignments of the common nodes must result in a larger matching size than the new assignments defined by q_c. Therefore, the algorithm adds the complement of the common nodes in V_{q_c} to each of the temporary maximal assignments (Algorithm 26, line 7-8).

Case (ii): if the stream item si_{q_i} has references to at least one stream item of q_c (Algorithm 25, line 14), then it means the edge (q_i, q_c) is a matching edge. Thus, we should combine the assignments of q_i with the assignments of q_c. This is described in Algorithm 27.

As we are looking for the maximal assignments for the subgraph rooted at q_i, it is sufficient to take only the maximal assignments of q_c into account. For that

Algorithm 27: Compute the combination of assignments

```
1  begin
2      let SI_{q_c} be a set of stream items referenced by si_{q_i};
3      m = max{si_{q_c}.maxMatchSize | si_{q_c} ∈ SI_{q_c}};
4      SI^{max}_{q_c} = {si_{q_c} | si_{q_c} ∈ SI_{q_c} ∧ si_{q_c}.maxMatchSize = m};
5      for each μ_{q_i} in tempMaxAssignmentSet[q_i, p_i] do
6          for each si_{q_c} in SI^{max}_{q_c} do
7              p_c = si_{q_c}.processNode;
8              for each μ_{q_c} in tempMaxAssignmentSet[q_c, p_c] do
9                  if commonNodes is empty then
10                     newMaxAssignments =
                       newMaxAssignments ∪ {μ_{q_i} ⊕ μ_{q_c}};
11                     newMatchSize = k1 * (m + 1) + k2 * |V^{match}_{μ_{qc}}|;
12                     si_{q_i}.maxMatchSize =
                       si_{q_i}.maxMatchSize + newMatchSize;
13                 else
14                     Combine assignments;
15     tempMaxAssignmentSet[q_i, p_i] = newMaxAssignments;
16 end
```

reason, we select all the stream items of q_c with the following properties: (i) stream items that are referenced by si_{q_i}; (ii) stream items that have the maximal matching size in the solution stream of q_c. These stream items are stored in the set $SI^{max}_{q_c}$ (Algorithm 27, line 4).

Next, the algorithm combines the existing assignments of q_i with the maximal assignments of q_c (Algorithm 27, line 5-14). The maximal assignments of q_c can be retrieved by iterating the assignments in *tempMaxAssignmentSet* that is associated with each stream item in $SI^{max}_{q_c}$ (line Algorithm 27, 6-8). With the matching process node p_c we can get $tempMaxAssignmentSet[q_c, p_c]$, which stores all the local maximal assignments for the subgraph rooted at q_c. The algorithm then iterates the assignments in $tempMaxAssignmentSet[q_c, p_c]$ (Algorithm 27, line 10). When combining the assignments, the algorithm examines whether there are possible overlaps of the subgraphs.

If there exists no common nodes between the subgraph rooted at q_c and the processed descendants of q_i, then we can simply combine the assignment μ_{q_i} with the assignment μ_{q_c}. The combination is kept in the set *newMaxAssignments* (Algorithm 27, line 10). The current maximal match size of q_i can be calculated as $maxMatchSize_{q_i} = maxMatchSize_{q_i} + m + 1$, where m is the match size of μ_{p_c} and 1 represents the new matching edge (q_i, q_c).

If there exist common nodes between the subgraph rooted at q_c and the processed descendants of q_i, we invoke Algorithm 28. In order to avoid overwriting the assignments of the processed descendants of q_i we remove the assignments of the common nodes from μ_{q_c} (Algorithm 28, line 2) and combine the remainder with μ_{q_i} (Algorithm 28, line 3). However, whether the new assignment is a maximal assignment of the subgraph rooted at q_i is unknown. We have to compare the matching size of the new assignment of q_i with the matching size of the original maximal assignments of q_i.

Algorithm 28: Combine assignments

```
1  begin
2      μ'_{q_c} = μ_{p_c} ⊖ commonNodes;
3      μ^{new}_{q_i} = μ_{q_i} ⊕ μ'_{q_c};
4      newMatchSize = k1 * |μ^{new}_{q_i}| + k2 * |V^{match}_{μ^{new}_{q_i}}|;
5      if newMatchSize > si_{q_i}.maxMatchSize then
6          newMaxAssignments = {μ^{new}_{q_i}};
7          si_{q_i}.maxMatchSize = newMatchSize;
8      else if newMatchSize = si_{q_i}.maxMatchSize then
9          newMaxAssignments = newMaxAssignments ∪ {μ^{new}_{q_i}};
10 end
```

If the match size of the new assignment is greater than that of the maximal assignment μ_{q_i} , then the new assignment is the maximal assignment. Thus, we set the match size of μ_{q_i} as the new maximal match size and replace all the assignments in the set *newMaxAssignments* with $\mu^{new}_{q_i}$ ((Algorithm 28, line 5-7).

If the match size of the new assignment equals to that of the maximal assignments μ_{q_i}, then the new assignment together with the assignments in *newMaxAssignments* are all the maximal assignments of the subgraph rooted at q_i (G_{q_i} for short). Thus, we add $\mu_{q_i}^{new}$ to the set *newMaxAssignments* ((Algorithm 28, line 8-9).

In both cases, the maximal match size of the stream item si_{q_i} is $k_1 * |\mu_{q_i}^{new}| + k_2 * V_{\mu_{q_i}^{new}}^{match}$.

Algorithm 29: Set the value of maxAssignmentSet

```
1 begin
2     if S_{q_i}.maxMatchSize < si_{q_i}.maxMatchSize then
3         S_{q_i}.maxMatchSize = si_{q_i}.maxMatchSize;
4         maxAssignmentSet[q_i] = tempMaxAssignmentSet[q_c, p_{q_c}];
5     else if S_{q_i}.maxMatchSize = si_{q_i}.maxMatchSize then
6         maxAssignmentSet[q_i] =
          maxAssignmentSet[q_i] ∪ tempMaxAssignmentSet[q_c, p_{q_c}];
7 end
```

At the end of the iteration of each stream item si_{q_i} in Algorithm 25, we update the maximal match size and the maximal assignment set associated with the solution stream S_{q_i} (Algorithm 29).

The worst case time complexity of the assignment phase is

$$O\left(|V_Q|(b+bc(c+n+n^2b))\right)$$

where $|V_Q|$ denotes the number of the nodes of the query graph Q, b is average number of the stream items of the query nodes, c is the average number of child nodes of the, n is the average number of assignments in *maxAssignmentSet*.

7.4.3 Combination Phase

Before we begin with the algorithm of the combination phase, let's first consider the example shown in Figure 7.1 again. In the assignment phase, we found two maximal level-0 assignments (global assignments) for the query graph, i.e. $(level-0)_1$ and $(level-0)_4$. Both of the maximal level-0 assignments has 7 matching edges and 7 matching nodes. Apparently, none of these maximal level-0 assignments reflect the actual similarity between the query graph and the process graph. The query graph and the process graph have an isomorphic subgraph rooted at c. But the mapping of the subgraph has not been included in any of the level-0 assignments. Thus, we cannot simply compute the actual similarity between the query graph and the process graph solely based on the maximal level-0 assignments. Instead, combining the maximal level-0 assignments with some maximal local assignments (maximal level-i assignments) could result in a more precise similarity value.

For example, we can combine the assignment $(level-0)_1$ with $(level-2)_1$, i.e. use the assignment for the query nodes $\{c,d,e,f,g,h\}$ in $(level-2)_1$ to substitute their assignments in $(level-0)_1$ as shown in Figure 7.7. The combination results in 10 matching edges and 11 matching nodes, which reflects the similarity between the query graph and the process graph more precisely. We use $U=\{\mu_o^{part},\mu_c^{part}\}$ to denote the combination of the assignments, where μ_o^{part} refers to the partial assignment that maps the query nodes $\{o,x,y,z,a,b\}$ and μ_c^{part} maps the rest of the query nodes $\{c,d,e,f,g,h\}$.

o	x	y	z	a	b	c	d	e	f	g	h
o_1	x_1	y_1	z_1	a_1	$\perp$	c_2	d_1	e_1	f_2	g_1	h_1
$(\text{Level-0})_1$						$(\text{Level-2})_1$					

Fig. 7.7 A maximal assignment for Q.

The combination phase tries to build global assignments that have the maximal similarity value by combining the maximal assignments found in the assignment phase. The basic idea is to successively use the maximal local as-

signments to substitute the assignments of the common nodes in the maximal global assignments. The substitution splits the original assignment that contains the common nodes into two partial assignments (cf. the example in Figure 7.7). We call an assignment that consists of more than one partial assignment a compound assignment. After the substitution, the algorithm compares the new similarity value with the previous similarity value. If the similarity after the substitution is greater than the previous similarity value, then the new compound assignment is defined to be the maximal global assignment. If the similarity after the substitution equals to the previous similarity value, then the new compound assignment is one of the maximal global assignments. Otherwise, the algorithm ignores the combination and continues with the examination until all maximal local assignments of the query nodes have been iterated. The procedures are shown in Algorithm 30.

At the beginning of the algorithm we store all the query nodes in the array $nodes[]$ sorted according to the following conditions: (i) it contains only query nodes where the $maxMatchSize$ of their solution streams is greater than 0; (ii) the query nodes are sorted in a descending order of the $maxMatchSize$ of their solution streams. Condition (i) lets the algorithm consider only query nodes whose maximal assignments may contribute to increase the overall similarity value. Condition (ii) ensures that the larger assignments are processed before the smaller ones (same motivation as in the assignment phase).

Then we initialize the similarity value s of the maximal level-0 assignments (root assignments). The similarity value can be calculated based on the maximal matching size stored with respective solution stream of the root q_r of the query graph (Algorithm 30, line 3-6). The combination starts with combining a maximal level-0 assignment with one of the biggest local assignments. If there is no maximal level-0 assignment in $maxAssignmentSet[q_r]$, we initialize an assignment $\mu_\perp$ to start with (Algorithm 30, line 7-8). The assignment $\mu_\perp$ maps all the query nodes in Q to $\perp$. M is used to store the compound assignments that resulted from the combination algorithm. The initial elements of M are the maximal level-0 assignments (Algorithm 30, line 9-10).

For each query node q_i in $nodes[]$ we iterate each global assignment U. As U could also be a compound assignment, we have to find the partial assign-

Algorithm 30: Combination Phase

begin
  $nodes[] = [q_0, q_1, q_2, ..., q_k]$;
  $M = \emptyset$;
  if $S_{q_r}.maxMatchSize \leq 0$ **then**
    $s = 0$;
  else
    $s = \frac{S_{q_r}.maxMatchSize}{k_1 * |E_Q| + k_2 * |V_Q^{match}|}$;
  if $maxAssignmentSet[q_r]$ *is empty* **then**
    $maxAssignmentSet[q_r] = \{\mu_\perp\}$;
  for *each* $\mu_{q_r} \in maxAssignmentSet[q_r]$ **do**
    $M = M \cup \{\mu_{q_r}\}$;
  if $nodes[]$ *is not empty* **then**
    for $i = 0$ *to* k **do**
      $M_{temp} = M$;
      for *each* $U \in M$ **do**
        let $\mu_{q_i}^{max} \in U$ be an assignment that includes q_i;
        $p_i = \mu^{max_{q_i}}(q_0)$;
        if $p_i = \perp$ *or* $si_{q_i}^{p_i}.maxMatchSize < S_{q_i}.maxMatchSize$ **then**
          $common = V_{\mu_{q_i}^{max}} \cap V_{q_i}$;
          $\mu_{q_i}^{'max} = \mu_{q_i}^{max} \ominus \mu_{q_i}^{common}$;
          for *each* $\mu_{q_i} \in maxAssignmentSet[q_i]$ **do**
            let μ'_{q_i} be the partial assignment of μ_{q_i} that only maps the common nodes;
            for *each* $\mu_{part} \in (U \backslash \mu_{q_i}^{max}) \cup \{\mu_{q_i}^{'max}\}$ **do**
              if $haveLostEdges(\mu_{part}, \mu'_{q_i})$ **then**
                $\mu''_{q_i} = \mu_{part} \oplus \mu'_{q_i}$;
                $U' = (U \setminus \{\mu_{part}, \mu_{q_i}^{common}\}) \cup \{\mu''_{q_i}\}$;
                $computeMaxAssignment(U', s)$;
              else
                if $\mu_{part} = \mu_{q_i}^{'max}$ **then**
                  $U' = (U \setminus \{\mu_{q_i}^{max}\}) \cup \{\mu_{q_i}^{'max}, \mu'_{q_i}\}$;
                  $computeMaxAssignment(U', s)$;
      $M = M_{temp}$;
end

ment $\mu_{q_i}^{max}$ in U that contains the query node q_i (Algorithm 30, line 15). The substitution is only meaningful when one of the following two conditions are satisfied (Algorithm 30, line 17): (i) the assignment $\mu_{q_i}^{max}$ maps q_i to undefined, i.e. $\perp$; (ii) the assignment $\mu_{q_i}^{max}$ maps q_i to a process node p_i and the maximal match size of the stream item $si_{q_i}^{p_i}$ is smaller than the maximal match size of the solution stream S_{q_i}.

Condition (i) means that the subgraph rooted at q_i has been assigned to $\perp$, which means its match size is 0. Substituting the assignment for the subgraph rooted at q_i would increase the overall similarity value. Condition (ii) indicates that the assignment associated with the stream item $si_{q_i}^{p_i}$ is not the stream item with the largest match size among other stream items in the solution stream S_{q_i}. So substituting the assignment for the subgraph rooted at q_i would also increase the overall similarity value. Otherwise, if the maximal match size of the stream item $si_{q_i}^{p_i}$ equals to the maximal match size of the solution stream S_{q_i}, then the partial assignment for the subgraph G_{q_i} is already a maximal assignment. Therefore, it is not necessary to substitute it with another maximal local assignment, as they have the same match size.

The next step is to identify the common nodes for the substitution. The common nodes is the intersection of the query nodes in $\mu^{max_{q_i}}$ and the query nodes of the subgraph G_{q_i} (Algorithm 30, line 18). For each maximal local assignment μ_{q_i} we are only interested in the partial assignment μ'_{q_i} that maps the common nodes (Algorithm 30, line 21). And for each partial assignment in U we are only interested in the partial assignment μ_{part} that does not map the common nodes (Algorithm 30, line 22). Now we try to combine the partial assignment μ'_{q_i} with the partial assignment μ_{part} and to examine whether the combination would result in a larger similarity value. However, when removing common nodes from the assignment $\mu_{q_i}^{max}$, we could have lost some matching edges. We call this the *lost-matching-edge problem*.

The lost-matching-edge problem is illustrated in Figure 7.8. In the figure we have a query graph Q and a process graph P. Nodes with the same index are the matching nodes. The maximal $level-0$ assignment is $\mu_{q_r} = \{[q_r/p_r][q_0/p_0][x/\perp][q_1/p_1][q_2/\perp][q_3/p_3][q_4/\perp][q_5/p_5]\}$. Note that the matching nodes of the subgraph rooted at q_2 are not in the $level-0$ assignments.

Thus, we want to combine the maximal $level-2$ assignment μ_{q_2} with μ_{q_r} and to examine whether the similarity is greater after the combination, where $\mu_{q_2} = \{[q_2/p_2][q_3/p_3][q_4/p_4][q_5/p_5]\}$. The assignments μ_{q_r} and μ_{q_2} have common nodes $\{q_2, q_3, q_4, q_5\}$. In order to combine the two assignments we subtract the mapping of the common nodes from μ_{q_r} and combine the mappings of the remaining nodes in μ_{q_r} with μ_{q_2} to form a compound assignment that maps all the nodes of the query graph. The compound assignment

$$\begin{aligned}\mu_{q_r}^{compound} &= \{\mu'_{q_r}, \mu_{q_2}\} \\ &= \{\{[q_r/p_r][q_0/p_0][x/\perp][q_1/p_1]\}, \{[q_2/p_2][q_3/p_3][q_4/p_4][q_5/p_5]\}\}.\end{aligned}$$

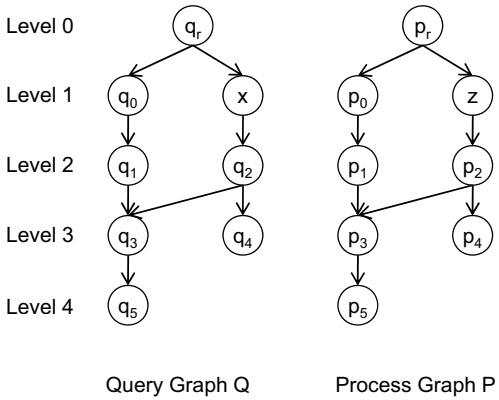


Fig. 7.8 Combine two assignments may lead to lost matching edges.

The number of matching edges of the compound assignment $\mu_{q_r}^{compound}$ is $|\mu'_{q_r}| + |\mu_{q_2}| = 2+3 = 5$. But from the example shown in Figure 7.8 we can see that the two graphs have actually 6 matching edges. The edge $e_{lost} = (q_1, q_3)$ is also a matching edge for the compound assignment, but getting lost after having split up the original assignment μ_{q_r}. This is the lost-matching-edge problem.

We can see from the example: if there exists at least one lost matching edge between two partial assignments, then we should not treat them as separate

assignments, but combine them to form one assignment as a whole. In the example above, we have two partial assignments μ'_{q_r} and μ_{q_2} in $\mu_{q_r}^{compound}$. As there exist one lost matching edge between μ'_{q_r} and μ_{q_2}, we combine them to form a larger single assignment, i.e.

$$\mu_{q_r}^{new} = \mu'_{q_r} \oplus \mu_{q_2} = \{[q_r/p_r][q_0/p_0][x/\perp][q_1/p_1][q_2/p_2][q_3/p_3][q_4/p_4][q_5/p_5]\}.$$

Algorithm 31: haveLostEdges(μ_1, μ_2)

```
1  begin
2      for each node q in V_{μ2} do
3          for each parent node q_parent of q do
4              if q_parent ∈ V_{μ1} then
5                  p1 = μ1(q_parent);
6                  p2 = μ2(q);
7                  if
                   ((p2 ≠ ⊥) ∧ there exists a reference from si_{q_parent}^{p1} to si_q^{p2})
                   then
8                      return true;
9          for each child node q_child of q do
10             if q_child ∈ V_{μ1} then
11                 p1 = μ1(q_child);
12                 p2 = μ2(q);
13                 if
                   ((p1 ≠ ⊥) ∧ there exists a reference from si_q^{p2} to si_{q_child}^{p1})
                   then
14                     return true;
15     return false;
16 end
```

In the combination phase we have to examine whether there exist lost matching edges between two partial assignments, so that we can decide whether to leave the assignments as separated or combine them into one (Algorithm 30, line 23). The lost matching edge problem occurs only if the start node of the edge is in one assignment and the end node of the edge is in another assignment.

Therefore, in Algorithm 31 we iterate the query nodes in one assignment and examine whether its parent or child node is in another assignment. As shown in Algorithm 31 we iterate the query nodes of μ_2 and examine the parent nodes of the query node q. If a parent node q_{parent} is in the assignment μ_1, then the edge $e = (q_{parent}, q)$ is a lost matching edge only if e is a matching edge. If there exists a reference from $si_{q_{parent}}$ to si_q, then the edge e is a matching edge, thus, also a lost matching edge. As soon as we found one lost matching edge, we can terminate the algorithm. Otherwise, the algorithm continues until all the query nodes are examined. Depending on the result of the Algorithm 31 we will (i) merge or (ii) split the assignments.

(i) If there exists a lost matching edge between μ'_{q_i} and any partial assignment in U, then we should merge the two assignments into one assignment (Algorithm 30, line 24). The the assignment of the common nodes μ'_{q_i} and the assignment μ_{part} in the original compound assignment U should be replaced by the merged assignment μ''_{q_i}. The new compound assignment U' is then the union of $U \setminus \left\{\mu_{part}, \mu_{q_i}^{common}\right\}$ and μ''_{q_i} (Algorithm 30, line 25).

We use the function $computeMaxAssignment\,(U', s)$ (Algorithm 32) to determine how we should deal with the new compound assignment U'. First, we need to determine whether U' delivers a larger similarity value (Algorithm 32, line 4-9). The compound assignment U' may contain more than one partial assignments. According to the Formula (7.3.3) we need to determine which partial assignment in U' is the biggest one. For that reason we iterate the partial assignments in the compound assignment and mark the biggest partial assignment with c_{max} (Algorithm 32, line 6-8). Then, we calculate the similarity using the Formula (7.3.3) (Algorithm 32, line 9). Next, based on the similarity value we can decide how to deal with the new compound assignment U'. If the similarity value s_{new} of U' is larger than the original one, then U' is the compound assignment that has the biggest similarity. Thus, we replace M_{temp} with $\{U'\}$. If s_{new} equals to s, then U' is one of the biggest compound assignments. Then, we add U' to M_{temp}. Otherwise, we do not achieve a larger similarity value through the combination and just drop U'.

(ii)If there exists no lost matching edges between $\mu_{q_i}^{'max}$ and μ'_{q_i} (Algorithm 30, line 27), then we can split the partial assignment $\mu_{q_i}^{max}$ into two parts: (a)

Algorithm 32: *computeMaxAssignment* (U', s)

```
1  begin
2      maxMatchSize = 0;
3      c_max = ∅;
4      for each assignment μ ∈ U' do
5          matchSize_μ = k1 * |μ| + k2 * |V_μ^match|;
6          if matchSize_μ > maxMatchSize then
7              maxMatchSize = matchSize_μ;
8              c_max = μ;
9      s_new = (k1*(|c_max| + k_structural * Σ{|μ'| | μ' ∈ U' \ {c_max}}) + k2*|V_U'^match|) / (k1*|E_Q| + k2*|V_Q|);
10     if s_new > s then
11         s = s_new;
12         M_temp = {U'};
13     else if s_new = s then
14         M_temp = M_temp ∪ {U'};
15 end
```

the mappings of the common nodes; (b) the mappings of the remaining nodes in $\mu_{q_i}^{max}$. Part (a) will be replaced by the partial assignment μ'_{q_i}, as the maximal match size of μ'_{q_i} is larger that that resulted through the mappings of the common nodes in $\mu_{q_i}^{'max}$ (Algorithm 30, line 17). Thus, we remove the partial assignment $\mu_{q_i}^{max}$ form U and use the partial assignments μ'_{q_i} (part (a)) and $\mu_{q_i}^{'max}$ (part (b)) (Algorithm 30, line 29). Also, here we have to examine whether U' has a larger similarity value than U (Algorithm 30, line 30).

And the end of each iteration of the query nodes the algorithm assigns M_{temp} to M (Algorithm 30, line 32).

The worst case time complexity of the combination phase is

$$O\left(m_1 + |V_Q| t n m_2 (l_1 p c (1 + m_2)\right),$$

where m_1 denotes the number of the maximal root assignments, t is the number of the maximal assignments in M, m_2 is the average number of partial assignments of maximal global assignments, l_1 is the average length of the partial assignments, p is the average number of the parent nodes of the query

nodes, c is the average number of the child nodes of the query nodes, n is the average number of assignments in *maxAssignmentSet* of all the query nodes.

Chapter 8
Architecture and Implementation

8.1 Introduction

This chapter presents the architecture and the implementation of an integrated modeling environment for supporting the design and reuse of BPEL fragments. The integrated modeling environment consists of a BPEL fragment editor and a BPEL repository. The BPEL fragment editor is a graphical editor for modeling BPEL processes and fragments. It extends the BPEL designer[1] with the support for modeling BPEL fragments and an extraction module that enables process modelers to extract BPEL fragments from existing BPEL process models. The BPEL repository is designed for storing BPEL process models and fragments. It provides the usual repository functionalities, such as versioning, locking, access control. It also integrates a query component that provides query capabilities for processing query requests on both the structure and the metadata of BPEL process models and fragments.

8.2 BPEL Fragment Editor

The BPEL fragment editor has been implemented based on the open source Eclipse BPEL Designer Project. The BPEL editor consists of an EMF data

[1] BPEL Designer Project, http://www.eclipse.org/bpel/

model that represents the XML schema of BPEL 2.0, a GEF-based graphical editor for modeling BPEL processes, a validator, a runtime framework that allows deployment and execution of BPEL processes on a BPEL engine, and a debug framework.

One of the major advantages of Eclipse platform is that it allows developers to extend the capabilities by installing Eclipse plug-ins. As the BPEL fragment editor aims at providing process modelers the capabilities of modeling both BPEL processes and BPEL fragments, we decide to implement the functionalities for modeling and extracting BPEL fragments as plug-ins. Our implementation extends the BPEL designer project with three subsystems:

- **de.uni.stuttgart.iaas.bpel.fragment.model:** implements the EMF model of the XML schema of BPEL fragment including utility functionalities such as serializer and deserializer;
- **de.uni.stuttgart.iaas.bpel.fragment.ui:** provides the graphical modeling elements and user interfaces for modeling BPEL fragments such as figures, palette, dialogs, and related actions for modeling operations;
- **de.uni.stuttgart.iaas.bpel.fragment.extraction:** provides capabilities for extracting BPEL fragments.

To model a BPEL fragment a process modeler can drag and drop the graphical modeling elements provided on the palette. The implementation extends the palette with additional modeling elements including $< fragment >$, $< bagActvity >$, and other BPEL 2.0 standard modeling constructs that have been promoted as first class modeling elements in the BPEL fragment schema (e.g. $< catch >$, $< catchAll >$, $< onEvent >$, etc.).

To extract a BPEL fragment a process modeler has to first select the activities and constructs that he/she wants to include in the resulting fragment. After having selected the respective elements on the canvas, the process modeler can add the selections to a Fragment Element Set. Then, the process modeler has to specify in which extraction mode the selected elements should be extracted. The extraction modes include the connected mode, the isolated mode, and the *connected (opaque)* mode. The connected (opaque) mode enables process modelers to use opaque activities for retaining the original process structure. During the extraction, elements that should be extracted in the connected

(opaque) mode will be extracted in the connected mode and will be replaced by $< opaqueActivity >$ after the extraction. The fragment element set stores both the selections and their respective extraction modes. Selections can be made in a successive manner for convenience. New Selections can be append to the fragment element set by repeating the add selection operation. Also process modelers can undo their added selections by selecting the elements and invoke the *remove activities from FragmentSet* in the popup context menu. Elements stored in the fragment selection set will be completely removed when the process modeler invokes *clear fragment set* or *extract fragment* from the context menu. *Show activities in FragmentSet* gives process modelers the possibility to gain an overview of their current constituents of the BPEL fragment being designed.

Extractions of BPEL fragments can be conducted in a successive manner. After extracting a BPEL fragment, a process modeler may want to add another part to the resulting BPEL fragment. Thus, the BPEL fragment editor provides a capability for appending a BPEL fragment to the previous extracted BPEL fragment. The successive extraction can also be used to avoid the effort on reducing a lot of generated opaque activities when there are a lot of not required intermediate activities lying between the required activities of a BPEL fragment. To append the extracted BPEL elements the process modeler has just to invoke the context menu *append fragment*.

Both *extract fragment* and *append fragment* trigger the extraction processing of the selected elements. Before the extraction the implementation examines whether the process modeler has selected a compound construct without having select any of its enclosed elements. In this case the implementation has to make sure whether the process modeler wants to extract the construct as an empty construct or with all its enclosed elements. A popup dialog will give the process modeler the possibility to refine the selection.

For selected elements that should be extracted in the connected mode, the implementation provides two options for the extraction. With the option *without reduction* the resulting BPEL fragments contain all generated opaque activities that are necessary for retaining the original control dependencies of the selected elements. The option *with reduction* results in a BPEL fragment whose

generated opaque activities have been removed with the help of the reduction algorithms that we have presented in the Section 5.4.

8.3 The BPEL Reuse Repository

The BPEL reuse repository (BPEL repository for short) enables process modelers to manage the reusable modeling artifacts. It is a shared database of modeling artifacts of BPEL processes and fragments as well as their associated metadata (cf. [30, 100, 148]). The architecture of the BPEL repository is shown in Figure 8.1.

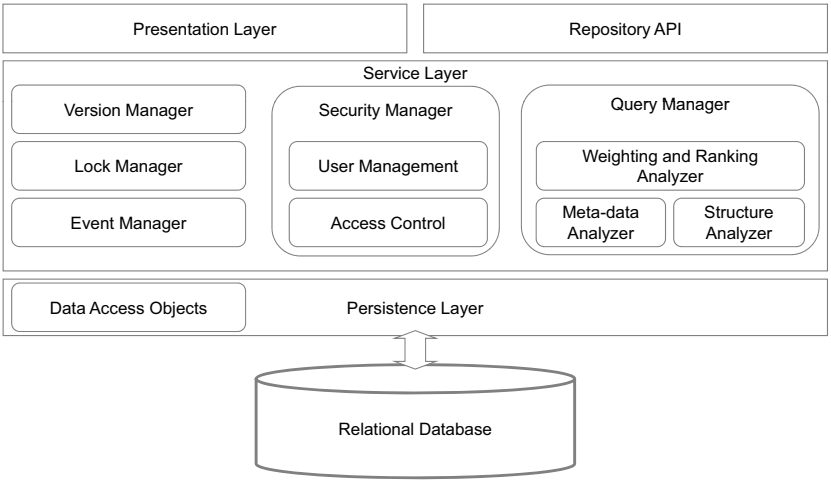


Fig. 8.1 The architecture of BPEL repository.

8.3.1 Presentation Layer

The presentation layer provides a Web interface, which enables process modelers to interact with the BPEL repository from a browser, such as searching for BPEL process models and fragments based on their meta data, checking out a BPEL process model or fragment, managing user accounts, viewing and releasing existing locks in the BPEL repository.

8.3.2 Repository API

Besides the Web Interface, the BPEL repository provides also a set of API to enable programmatic access to the repository. It can be used to integrate tools that interact with the BPEL repository, e.g. the BPEL fragment editor. The repository API provides the functions such as $create$, $check-in$, $check-out$, $import$, $export$, $lock$, $unlock$, and also functions for user management. Details about the repository API can be found in [152].

8.3.3 Service Layer

The service layers compasses functionalities for storing, retrieving, and managing the modeling artifacts and their associated metadata stored in the BPEL repository. The BPEL repository uses five service managers to implement these functionalities.

8.3.3.1 Version Manager

Typically, a BPEL process model or a BPEL fragment undergoes a series of revisions during the modeling phase. Each can be captured as a version. A version is a snapshot of the BPEL process model or BPEL fragment being modeled at some point in its modeling lifecycle [30]. The version manager provides

functionalities for capturing the version history of a modeling artifact. It coordinates with the locking manager to set and release a lock when checking out and checking in modeling artifacts. Modeling artifacts in the BPEL repository are not isolated. A BPEL process model may reuse one or more BPEL fragments. The version mananger also captures the reuse relationships between the BPEL process model and the BPEL fragments.

8.3.3.2 Lock Manager

The lock manager is responsible for setting and releasing locks on modeling artifacts. The BPEL repository implements a pessimistic locking mechanism. It means that a modeling artifact is exclusively locked for the process modeler who checked it out. If there exists already a lock on it, then the check-out can only be conducted after the lock has been released. The releasing of a lock can either be trigger by checking in the modeling artifact again by the process modeler who checked out the modeling artifact before or be released by the repository administrator manually.

8.3.3.3 Security Manager

In order to prevent unauthorized access to the BPEL repository, each user has to authenticate herself/himself before interacting with the BPEL repository. The security manager of the BPEL repository comprises of two logical components: user manager and access control. User manager implements three different roles: administrator, designer, and basic user. A basic user can only search and export modeling artifacts. A designer can also check out and check in modeling artifacts. Administrators have full access to the BPEL repository. In addition, they can also interact with the locking manager to release existing locks manually and manage user accounts and rights of the BPEL repository. Access control implements the functionalities for assigning the appropriate rights to each user account and the authentication mechanism. The functionalities of the secu-

rity manager have been implemented by using the Spring security framework [2].

8.3.3.4 Query Manager

The query manager comprises of three logical components: structure analyzer, metadata analyzer, and the weighting and ranking analyzer.

The metadata analyzer implements the query mechanism for the metadata that are associated with the modeling artifacts in the BPEL repository. The metadata analyzer has been implemented with Apache Lucene, which is a search engine library written in Java. Lucene brings several convenient features, such as an algorithm for ranking the query results based on their relevance to the query phrases, pagination of query results, i.e. query results are not returned all at once, but page by page.

The structure analyzer implements the query algorithm that we have discussed in Chapter 7. It takes XML-representation of a BPEL fragment or process model as the query request and transforms it into the graph representation that we discussed in Chapter 6. As the graph matchmaking of the query request with all the BPEL fragments and process models in the repository could be very expensive, the structure analyzer invokes the metadata analyzer to filter BPEL fragments and process models that should be considered in the structural matchmaking, if the metadata are available in the query request.

After the query processing, the weighting and ranking analyzer will rank the query results based on their values in metadata similarity and structural similarity. The weighting factors can be set when configuring the repository.

8.3.4 Persistence Layer

The persistence layer uses Data Access Object (DAO), which provides an abstract interface to storage mechanisms of BPEL fragments and process mod-

[2] http://static.springsource.org/spring-security/site/

els. As BPEL fragments and process models are specified using a XML-representation, it would be convenient to manipulate and access them in an object model. We use JAXB map the XML schema to object classes and Hibernate to map object classes to database tables in a relational database, e.g. MySQL.

Chapter 9
Conclusion and Outlook

In this thesis we introduced BPEL fragments to enhance reuse of process logic in BPEL process models. In Chapter 1 we discussed the motivation of our research work. Reuse is an established concept to improve productivity and quality. Today, subprocesses represent the only granule of reuse. A subprocess is either a self-contained business process invoked by another business process or an inline process that must be defined in a scope activity. In both cases, subprocesses do not allow process modelers to reuse arbitrary parts of a business process, especially parts that cannot be seen as self-contained business processes and parts that need to be reused within other BPEL constructs rather than a scope activity.

Based on this motivation we studied the related work in Chapter 2. Besides subprocesses we discussed related approaches for reusing process models as a whole. After that we gave an overview of decomposing process models into process fragments. Especially, we analyzed approaches of decomposing BPEL process models for distributed execution, for analyzing purpose, for generating process views, etc. A reuse approach should allow users easily finding reusable modeling artifacts. Thus, we took a look on existing query approaches for process models. One key procedure in these approaches is similarity measurement, which is also a part of the related work that we discussed at the end of Chapter 2.

In Chapter 3 we introduced the concept of process fragments for reuse. First we defined process fragments from the process point of view. Second, we also presented a formal definition of process fragments from the graph point of view, in order to use graph-based query algorithms later. Based on the graph view we have identified 4 different shapes of process fragments according to the numbers of their entry and exit nodes. The discussion on granularity helps us to understand in which extend a process model can be broken down into process fragments. And different reuse styles specify to which extend a process fragment can be modified at the time of reuse. Last but not least, the lifecycle of process fragments refines the process design phase in the conventional business process management lifecycle. We outlined the phases of the lifecycle that are addressed in this thesis.

In Chapter 4 we presented the BPEL fragment modeling language. The chapter began with the requirement analysis, which considered characteristics in the definition of process fragments from the process view, such as syntactical and semantic incompleteness. In addition, the analysis took also main possible types of BPEL fragments into account. One of the design goals is to reuse as much modeling constructs of BPEL as possible. Thus, we also evaluated whether executable BPEL or abstract BPEL should be used as the basis for designing BPEL fragment modeling language. BPEL fragment modeling language also introduced `<bagActivity>`. Similar to opaque activities, the bag activity enables process modelers to model unknown process logic. Different from opaque activities, a bag activity can be completed either by exactly one activity or by a BPEL fragment.

Reusable process logic can be either modeled from scratch or be extracted from existing BPEL process models. In Chapter 5 we introduced a mechanism for extracting BPEL fragments from BPEL process models. The extracting mechanism comprises three phases: the selection phase, the construction phase, and the reduction phase. In the selection phase process modelers manually select process activities for extraction and specify whether the original control dependencies should be retained after the extraction. In the construction phase, our approach extracts the Lowest Common Nesting Ancestor (LCNC) of all the selected activities and replaces the not selected activities with opaque activ-

ities that are generated by our algorithm. The generated opaque activities can be considered as redundant. Thus, we defined reduction rules for removing the generated opaque activities while still retaining the original control dependencies of the remaining activities. Also we presented the reduction algorithms that are designed according on the reduction rules.

In order to use graph-based algorithms we describe in Chapter 6 a mapping framework to map a BPEL process model or fragment to a directed and acyclic graph. The mapping framework extends the one proposed by Khalaf [78] to meet the requirements that we have identified in our research.

Chapter 7 presented the graph-based query algorithm. The query algorithm returns not only BPEL process models and fragments that satisfy exactly the query request, but are also able to find process models and fragments that partially match the query request. The query algorithm consists of three phases: the initialization phase, the assignment phase, and the combination phase. In the initialization phase the algorithm compares each node with the nodes in the process graph. For each matching node a stream item is created and associated with the corresponding query node. Based on the data generated in the initialization phase, the algorithm computes for each subgraph of the query graph the maximal similarity in the assignment phase. In the combination phase the algorithm computes the maximal similarity between the process graph and the query graph by combining the subgraphs that found in the assignment phase.

As proof of concept we described in Chapter 8 the prototypical implementation of the concepts discussed in this thesis. The BPEL process designer demonstrates a modeling tool that enables process modelers to extract BPEL fragments from existing BPEL process models. A reuse repository allows process modelers to store BPEL process models and fragments. It also implements the query algorithm that we discussed in Chapter 7. We also discussed the architecture of the implementations and the key components.

Outlook

During the course of the thesis we have identified several topics for future work.

Process fragment profile: similar to abstract process profiles, a BPEL fragment profile should specify how the BPEL fragment language can be used. For example, the profile could specify reuse styles for each element in the BPEL fragment, completion rules for opaque and bag activities, resolving non-unified links, etc.

Selecting activities through SQL-query: to extract a BPEL fragment process modelers have to manually select the activities in a BPEL process model for extraction. If the selection contains a large amount of activities then the manual selection could be cumbersome and error-prone. In this case, selection using SQL-like queries could be helpful. The required activities may not share common attributes so that they can be selected at once by one query. Thus, the modeling tool should enable process modelers to successively select the required activities through more that one query. Also when making a complete selection, the enclosed activities inherit the extraction mode of the enclosing activity. A more flexible inheritance strategy is, for example, to inherit the extraction mode to level k. From the level $k+1$ process modelers can assign another extraction mode to the elements.

Transition and join conditions during extraction: In this thesis, transition and join conditions are removed during the extraction. However, they could be useful for process modelers to understand the operational semantics of the BPEL fragment. Also, transition and join conditions could be reused in modeling new process models.

Duplicated handlers during extraction: when extracting fault handlers the extraction algorithm should examine whether the fault handlers already exists in the BPEL fragment, e.g. through comparing qualified fault names. However, this requires domain-specific knowledge to be able to judging whether the two fault handlers have the same operational semantics.

Query with incomplete links and bag activities: the query mechanism in this thesis does not take incomplete links into consideration. BPEL fragments with incomplete links could be also needed, especially when composing with other BPEL fragments as presented in [48]. Also we did not consider bag activities in the query algorithm in this thesis. If the process graph contains a bag activity, it can match to every activity in the query request, and vice versa. An intelligent

matchmaking method for bag activities that reduces the number of matching activities is needed.

Composition of BPEL fragments: the composition phase in the process fragment lifecycle embodies the value of reuse of process fragments. This phase has not been addressed in this thesis. However, Eberle [48] has conducted research work in that area. The open questions include: how to match the exit points of the preceding fragment with the entry points of the succeeding fragment; how to validate the control flow and data flow after composition, etc.

References

1. Collins Dictionary. URL http://www.collinslanguage.com
2. Merriam-Webster's Online Dictionary. URL http://www.merriam-webster.com
3. Workflow management coalition terminology & glossary. Tech. rep., Workflow Management Coalition (1999)
4. Business Process Modeling Notation (BPMN) Specification, version 1.1. The Object Management Group (OMG) (2008). URL http://www.omg.org/spec/BPMN/1.1/PDF
5. van der Aalst, W.M.P.: Business process management: A personal view. Business Process Management Journal **10**(2) (2004). URL http://wwwis.win.tue.nl/~wvdaalst/publications/p229.pdf
6. Abiteboul, S., Benjelloun, O., Milo, T.: The active XML project: an overview. The VLDB Journal **17**(5), 1019–1040 (2008). DOI http://dx.doi.org/10.1007/s00778-007-0049-y
7. Ackermann, J., Brinkop, F., Conrad, S., Fettke, P., Frick, A., Glistau, E., Jaekel, H., Kotlar, O., Loos, P., Mrech, H., Ortner, E., Raape, U., Overhage, S., Sahm, S., Schmietendorf, A., Teschke, T., Turowski, K.: Standardized specification of business components - memorandum of the working group 5.10.3 - component oriented business application system (2002)
8. Adams, M., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Facilitating flexibility and dynamic exception handling in workflows through worklets. In: Proceedings 17th International Conference on Advanced Information Systems Engineering (CAiSE 2005), pp. 45–50. Springer Verlag (2005). URL http://sky.fit.qut.edu.au/~adamsmj/05Worklets_CaiseForum.pdf
9. Adams, M., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Worklets: A service-oriented implementation of dynamic flexibility in workflows. In: Proceed-

ings the 14th International Conference on Cooperative Information Systems (CoopIS 2006), vol. 4275, pp. 291–308. Springer Verlag (2006). DOI http://dx.doi.org/10.1007/11914853_18

10. Agnarsson, G., Greenlaw, R.: Graph Theory: Modeling, Applications, and Algorithms. Prentice Hall (2007)
11. Agrawal, A., Amend, M., Das, M., Ford, M., Keller, C., Kloppmann, M., König, D., Leymann, F., Müller, R., Pfau, G., Plösser, K., Rangaswamy, R., Rickayzen, A., Rowley, M., Schmidt, P., Trickovic, I., Yiu, A., Zeller, M.: WS-BPEL extension for people (BPEL4People). Tech. rep., OASIS WS-BPEL Extension for People (BPEL4People) Technical Committee (2007). URL `http://xml.coverpages.org/bpel4people.html`
12. Akutsu, T., Fukagawa, D., Takasu, A.: Improved approximation of the largest common subtree of two unordered trees of bounded height. Information Processing Letters **109**(2), 165–170 (2008). DOI http://dx.doi.org/10.1016/j.ipl.2008.09.025
13. Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M., Srivastava, D., Wu, Y.: Structural joins: A primitive for efficient XML query pattern matching. In: Proceedings of the 18th International Conference on Data Engineering, pp. 141–152. IEEE Computer Society (2002)
14. Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S.: A Pattern Language: Towns, Buildings, Construction. Oxford University Press (1977)
15. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures and Applications. Springer Verlag (2004)
16. Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guíar, A., Kartha, N., Liu, C.K., Khalaf, R., König, D., Marin, M., Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P., Yiu, A.: Web Services Business Process Execution Language 2.0. Organization for the Advancement of Structured Information Standards (OASIS) (2007). URL `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`
17. Ambler, S.: A realistic look at object-oriented reuse. Software Development **6**(1), 30–38 (1998)
18. Awad, A.: BPMN-Q: A language to query business processes. In: Proceedings of the 2nd International Workshop on Enterprise Modelling and Information Systems Architectures (EMISA 2007), *LNI*, vol. P-119, pp. 115–128. GI (2007)
19. Awad, A., Decker, G., Weske, M.: Efficient compliance checking using BPMN-Q and temporal logic. In: Proceedings of the 6th International Conference on Business Process Management (BPM 2008). Springer (2008). DOI http://dx.doi.org/10.1007/978-3-540-85758-7_24
20. Awad, A., Polyvyanyy, A., Weske, M.: Semantic querying of business process models. In: Proceedings of the 12th International Conference on Enterprise Distributed Object Computing (EDOC 2008) (2008)

21. Bajaj, S., Box, D., Chappell, D., Curbera, F., Daniels, G., Hallam-Baker, P., Hondo, M., Kaler, C., Langworthy, D., Nadalin, A., Nagaratnam, N., Prafullchandra, H., von Riegen, C., Roth, D., Schlimmer, J., Sharp, C., Shewchuk, J., Vedamuthu, A., Yalcinalp, Ü., Orchard, D.: Web Services Policy 1.2 - Framework (WS-Policy). W3C (2006). URL http://www.w3.org/Submission/2006/SUBM-WS-Policy-20060425/
22. Basili, V.R., Briand, L.C., Melo, W.L.: How reuse influences productivity in object-oriented systems. Communications of the ACM **39**(10), 104–116 (1996). DOI http://dx.doi.org/10.1145/236156.236184
23. Becker, J.: Lexikon der Wirtschafisinformatik, chap. Referenzmodel, pp. 399–400 (2001). (In German)
24. Becker, J., Delfmann, P., Dreiling, A., Knackstedt, R., Kuropka, D.: Configurative process modeling-outlining an approach to increased business process model usability. In: Proceedings of the 14th Information Resources Management Association International Conference, pp. 615–619. IRM Press (2004)
25. Becker, J., Kugeler, M., Rosemann, M.: Prozessmanagement Ein Leitfaden zur prozessorientierten Organisationsgestaltung, 5 edn. Springer Verlag (2005). (In German)
26. Beeri, C., Eyal, A., Kamenkovich, S., Milo, T.: Querying business processes with BP-QL. In: Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005), pp. 1255–1258. VLDB Endowment (2005)
27. Beeri, C., Eyal, A., Kamenkovich, S., Milo, T.: Querying business processes. In: Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB 2006), pp. 343–354. VLDB Endowment (2006)
28. Beeria, C., Eyalb, A., Kamenkovichb, S., Milob, T.: Querying business processes with BP-QL. Information Systems **33**(6), 477–507 (2008). DOI http://dx.doi.org/10.1016/j.is.2008.02.005
29. Berge, C.: Isomorphism problems for hypergraphs. In: Hypergraph Seminar, vol. 411, pp. 1–12 (1974). DOI http://dx.doi.org/10.1007/BFb0066174
30. Bernstein, P.A., Dayal, U.: An overview of repository technology. In: Proceedings of the 20th International Conference on Very Large Data Bases (VLDB), pp. 705–713. Morgan Kaufmann Publishers Inc. (1994). URL http://www.vldb.org/conf/1994/P705.PDF
31. Bille, P.: A survey on tree edit distance and related problems. Theoretical Computer Science **337**(1-3), 217–239 (2005). DOI http://dx.doi.org/10.1016/j.tcs.2004.12.030
32. Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J.H., Pohl, K.: Variability issues in software product lines. In: Proceeding of the 4th International Workshop Software Product-Family Engineering (PFE), vol. 2290, pp. 13–21. Springer Verlag (2002). DOI http://dx.doi.org/10.1007/3-540-47833-7_3
33. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: Proceedings of the 2002 ACM SIGMOD International Conference

on Management of Data (SIGMOD 2002), pp. 310–321. ACM (2002). DOI http://doi.acm.org/10.1145/564691.564727

34. Bunke, H.: On a relation between graph edit distance and maximum common subgraph. Pattern Recognization Letter **18**(9), 689–694 (1997). DOI http://dx.doi.org/10.1016/S0167-8655(97)00060-3
35. Bunke, H., Kandel, A.: Mean and maximum common subgraph of two graphs. Pattern Recognition Letters **21**(2), 163–168 (2000). DOI http://dx.doi.org/10.1016/S0167-8655(99)00143-9
36. Bunke, H., Shearer, K.: A graph distance metric based on the maximal common subgraph. Pattern Recognition Letters **19**(3-4), 255–259 (1998). DOI http://dx.doi.org/10.1016/S0167-8655(97)00179-7
37. Buschmann, F., Henney, K., Schmidt, D.C.: Past, present, and future trends in software patterns. IEEE Software **24**(4), 31–37 (2007). DOI http://doi.ieeecomputersociety.org/10.1109/MS.2007.115
38. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture Volume 1: A System of Patterns, 1 edn. Wiley (1996)
39. Card, D.: Why do so many reuse programs fail? IEEE Software **11**(5), 114–115 (1994). DOI http://dx.doi.org/10.1109/52.311078
40. Chen, L., Gupta, A., Kurul, M.E.: Stack-based algorithms for pattern matching on DAGs. In: Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005), pp. 493–504. VLDB Endowment (2005). URL `http://www.vldb2005.org/program/paper/wed/p493-chen.pdf`
41. Chien, S.Y., Vagena, Z., Zhang, D., Tsotras, V.J., Zaniolo, C.: Efficient structural joins on indexed XML documents. In: Proceedings of the 28th international conference on Very Large Data Bases, VLDB '02, pp. 263–274. VLDB Endowment (2002). URL `http://portal.acm.org/citation.cfm?id=1287369.1287393`
42. Chinosi, M.: Representing business process : Conceptual model and design methodology. Ph.D. thesis, Universitá degli studi dell'Insubria (2008)
43. Corrales, J.C., Grigori, D., Bouzeghoub, M.: BPEL processes matchmaking for service discovery. In: Proceedings of OTM Confederated International Conferences on the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, vol. 4275, pp. 237–254. Springer (2006). DOI http://dx.doi.org/10.1007/11914853_15
44. Corrales, J.C., Grigori, D., Bouzeghoub, M., Burbano, J.E.: Bematch: a platform for matchmaking service behavior models. In: Proceedings of the 11th International Conference on Extending Database Technology (EDBT 2008), pp. 695–699. ACM (2008). DOI http://doi.acm.org/10.1145/1353343.1353428
45. Danylevych, O., Karastoyanova, D., Leymann, F.: Optimal stratification of transactions. In: Proceedings of the Fourth International Conference on Internet and Web Applications and Services (ICIW) (2009)
46. Deutch, D., Milo, T.: Querying structural and behavioral properties of business processes. In: Proceedings of the 11th International Symposium on Database Program-

ming Languages (DBPL 2007), *Lecture Notes in Computer Science*, vol. 4797, pp. 169–185. Springer (2007)

47. Dongen, B., Dijkman, R., Mendling, J.: Measuring similarity between business process models. In: Proceedings of the 20th international conference on Advanced Information Systems Engineering, CAiSE '08, pp. 450–464. Springer Verlag (2008). DOI http://dx.doi.org/10.1007/978-3-540-69534-9_34. URL http://dx.doi.org/10.1007/978-3-540-69534-9_34
48. Eberle, H., Leymann, F., Schleicher, D., Schumm, D., Unger, T.: Process fragment composition operations. In: Proceedings of the 2010 IEEE Asia-Pacific Services Computing Conference, APSCC '10, pp. 157–163. IEEE Computer Society (2010). DOI http://dx.doi.org/10.1109/APSCC.2010.72
49. Eberle, H., Unger, T., Leymann, F.: Process fragments. In: Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on the Move to Meaningful Internet Systems: Part I, OTM '09, pp. 398–405. Springer-Verlag (2009). DOI http://dx.doi.org/10.1007/978-3-642-05148-7_29
50. Erl, T.: SOA Principles of Service Design. Prentice Hall (2007)
51. Eshuis, R., Grefen, P.: Constructing customized process views. Data & Knowledge Engineering **64**(2), 419–438 (2007). DOI http://dx.doi.org/10.1016/j.datak.2007.07.003
52. Fellner, K.J., Turowski, K.: Classification framework for business components. In: Proceedings of the 33rd Annual Hawaii International Conference on System Sciences, pp. 10–19. IEEE Computer Society (2000). DOI http://dx.doi.org/10.1109/HICSS.2000.927009
53. Ferrer, M., Valveny, E., Serratosa, F.: Median graph: A new exact algorithm using a distance based on the maximum common subgraph. Pattern Recognition Letters **30**(5), 579–588 (2009). DOI http://dx.doi.org/10.1016/j.patrec.2008.12.014
54. Fettke, P., Loos, P.: Classification of reference models: a methodology and its application. Journal of Information Systems and E-business Management **1**, 35–53 (2003). DOI 10.1007/BF02683509
55. Fettke, P., Loos, P.: Using reference models for business engineering - state-of-the-art and future developments. In: Proceedings of Innovations in Information Technology, pp. 1–5 (2006). DOI http://dx.doi.org/10.1109/INNOVATIONS.2006.301959
56. Fettke, P., Loos, P., Zwicker, J.: Business process reference models: Survey and classification. In: Proceedings of Business Process Management Workshops, vol. 3812/2006, pp. 469–483. Springer Verlag (2006). DOI http://dx.doi.org/10.1007/11678564_44
57. Filipowska, A., Hepp, M., Kaczmarek, M., Kowalkiewicz, M., Markovic, I., Starzecka, M., Stolarski, P., Todorova, P., Walczak, A., Zhou, X.: Deliverable 1.2 - business process oriented organizational ontology. Tech. rep., Project IST 026850 - Semantics Utilized for Process management within and between Enterprises (SUPER) (2007). URL http://www.ip-super.org/res/Deliverables/M18/D1.2.pdf

58. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Professional (2002)
59. Frakes, W.B., Fox, C.J.: Quality improvement using a software reuse failure modes model. IEEE Transactions on Software Engineering **22**(4), 274–279 (1996). DOI http://dx.doi.org/10.1109/32.491652
60. G. Keller, M.N., Scheer, A.: Semantische Prozessmodellierung auf der Grundlage Ereignisgesteuerter Prozessketten (EPK). Tech. Rep. Heft 89, Instituts für Wirtschaftsinformatik, University of Saarland (1992). (In German)
61. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software, 1 edn. Addison-Wesley (1994)
62. Gerlach, D.: Mining of business process fragments. Master's thesis, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany (2008). URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2732&engl=0
63. Goderis, A., Li, P., Goble, C.: Workflow discovery: The problem, a case study from e-science and a graph-based solution. In: Proceedings of the IEEE International Conference on Web Services (ICWS), pp. 312–319. IEEE Computer Society (2006). DOI http://dx.doi.org/10.1109/ICWS.2006.147
64. Görlach, K., Kopp, O., Leymann, F., Schumm, D., Strauch, S.: WS-BPEL Extension for Compliance Fragments (BPEL4CFrags), Version 1.0. Technischer Bericht Informatik 2011/01, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany (2011). URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2011-01&engl=
65. Gottschalk, F., van der Aalst, W., Jansen-Vullers, M.: Configurable process models: A foundational approach. Reference Modeling - Efficient Information Systems Design Through Reuse of Information Models, pp. 59–78 (2007). DOI http://dx.doi.org/10.1007/978-3-7908-1966-3_3
66. Gottschalk, F., Rosemann, M., van der Aalst, W.: My own process: Providing dedicated views on EPCs. EPK pp. 156–175 (2005). (In German)
67. Gschwind, T., Koehler, J., Wong, J.: Applying patterns during business process modeling. In: Proceedings of the 6th International Conference on Business Process Management (BPM), pp. 4–19. Springer Verlag (2008). DOI http://dx.doi.org/10.1007/978-3-540-85758-7_4
68. Haas, H., Brown, A.: Web services glossary. Tech. rep., The World Wide Web Consortium (W3C) (2004). URL http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/
69. Hammer, M., Champy, J.: Reengineering the Corporation: A Manifesto for Business Revolution. HarperBusiness (1994)
70. Hassine, I., Rieu, D., Bounaas, F., Seghrouchni, O.: Towards a reusable business components model. In: Proceedings of Workshop on Reuse in Object-oriented Informa-

tion Systems Design in conjunction with the 8th International Conference on Object-Oriented Information Systems (OOIS 2002) (2002). URL http://www-lsr.imag.fr/OOIS_Reuse_Workshop/Papers/Hassine.pdf

71. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional (2003)
72. Hwang, F.K., Richards, D.S., Winter, P.: The Steiner Tree Problem (Annals of Discrete Mathematics). North-Holland (1992)
73. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley (1997)
74. Kann, V.: On the approximability of the maximum common subgraph problem. In: Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science (STACS), pp. 377–388. Springer-Verlag (1992)
75. Karastoyanova, D.: Enhancing flexibility and reusability of Web service flows through parameterization. Ph.D. thesis, Fachbereich Informatik Technische Universität Darmstadt (2006)
76. Keller, G., Teufel, T.: SAP R/3 Process Oriented Implementation - Iterative Process Prototyping. Addison-Wesley (1998)
77. Khalaf, R.: From Rosettanet pips to BPEL processes: A three level approach for business protocols. Data Knowledge Engineering **61**(1), 23–38 (2007). DOI http://dx.doi.org/10.1016/j.datak.2006.04.006
78. Khalaf, R.: Supporting business process fragmentation while maintaining operational semantics: A BPEL perspective. Ph.D. thesis, Institut für Architektur von Anwendungssystemen der Universität Stuttgart (2008). URL http://elib.uni-stuttgart.de/opus/volltexte/2008/3514/
79. Khalaf, R., Leymann, F.: Role-based decomposition of business processes using BPEL. In: Proceedings of the IEEE International Conference on Web Services (ICWS 2006), pp. 770–780. IEEE Computer Society (2006). DOI http://dx.doi.org/10.1109/ICWS.2006.56
80. Khalaf, R., Leymann, F.: Coordination for fragmented loops and scopes in a distributed business process. In: 8th International Conference on Business Process Management (BPM 2010). Springer-Verlag (2010). URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2010-42&engl=
81. Kim, K.H., Won, J.K., Kim, C.M.: A fragment-driven process modeling methodology. In: Proceedings of International Conference on Computational Science and Its Applications (ICCSA 2005), vol. 3482/2005, pp. 817–826. Springer Verlag (2005). DOI http://dx.doi.org/10.1007/11424857_89
82. Kloppmann, M., Koenig, D., Leymann, F., Pfau, G., Rickayzen, A., von Riegen, C., Schmidt, P., Trickovic, I.: WS-BPEL extension for subprocesses BPEL-SPE - a joint white paper by IBM and SAP (2005). URL

http://download.boulder.ibm.com/ibmdl/pub/software/dw/webservices/ws-bpelsubproc/ws-bpelsubproc.pdf

83. Kopp, O., Eberle, H., Leymann, F., Unger, T.: The Subprocess Spectrum. In: Proceedings of the Business Process and Services Computing Conference: BPSC 2010, *Lecture Notes in Informatics*, vol. P-177, pp. 267–279. Gesellschaft für Informatik e.V. (GI) (2010). URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2010-73&engl=
84. Koschmider, A.: Ähnlichkeitsbasierte Modellierungsunterstützung für Geschäftsprozesse. Ph.D. thesis, Fakultät für Wirtschaftswissenschaften der Universität Fridericiana zu Karlsruhe (2007). URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000007200
85. Krueger, C.W.: Software reuse. ACM Computing Surveys (CSUR) **24**(2), 131–183 (1992). DOI http://dx.doi.org/10.1145/130844.130856
86. Kuramochi, M., Karypis, G.: Frequent subgraph discovery. Journal of ICDM pp. 313–320 (2001)
87. Kuropka, D.: Modelle zur Repräsentation natürlichsprachlicher Dokumente. Logos Verlag (2003)
88. La Rosa, M., Dumas, M., ter Hofstede, A.H.M., Mendling, J.: Configurable multi-perspective business process models. Journal of Information Systems **36**, 313–340 (2011). DOI http://dx.doi.org/10.1016/j.is.2010.07.001
89. Lam, W.: Process reuse using a template approach: a case-study from avionics. ACM SIGSOFT Software Engineering Notes **22**(2), 35–38 (1997). DOI http://doi.acm.org/10.1145/251880.251924
90. Lau, J.M., Iochpe, C., Thom, L.H., Reichert, M.: Discovery and analysis of activity pattern co-occurrences in business process models. In: Procedings of 11th International Conference on Enterprise Information Systems (ICEIS'09), pp. 83–88 (2009). URL http://dblp.uni-trier.de/db/conf/iceis/iceis2009-3.html#LauITR09
91. Lau, K.K., Wang, Z.: Software component models. IEEE Transactions on Software Engineering **33**(10), 709–724 (2007). DOI http://dx.doi.org/10.1109/TSE.2007.70726
92. Levi, G.: A note on the derivation of maximal common subgraphs of two directed or undirected graphs. Calcolo **9**(4), 341–352 (1973)
93. Leymann, F., Altenhuber, W.: Managing business processes as an information resource. IBM System Journal **33**(2), 326–348 (1994). URL http://www.research.ibm.com/journal/sj/332/ibmsj3302H.pdf
94. Leymann, F., Karastoyanova, D., Papazoglou, M.: Business Process Management Standards. Handbook on Business Process Management 1. Springer-Verlag (2010). URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INBOOK-2010-02&engl=
95. Leymann, F., Roller, D.: Production Workflow - Concepts and Techniques. Prentice Hall (2000)

96. Leymann, F., Roller, D.: Modeling business processes with BPEL4WS. Information Systems and E-Business Management **4**(3), 265–284 (2005). DOI http://dx.doi.org/10.1007/s10257-005-0025-2
97. Lim, W.C.: Effects of reuse on quality, productivity, and economics. IEEE Software **11**(5), 23–30 (1994). DOI http://dx.doi.org/10.1109/52.311048
98. Lindert, F., Deiters, W.: Modelling inter-organizational processes with process model fragments. In: Proceedings of Workshop Informatik '99: Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications, vol. 24, pp. 33–41. CEUR Workshop Proceedings (1999). URL http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-24/ld99.pdf
99. Ma, Z., Leymann, F.: A lifecycle model for using process fragment in business process modeling. In: Proceedings of the 9th Workshop on Business Process Modeling, Development, and Support (BPDMS), pp. 1–9 (2008). URL http://lams.epfl.ch/conference/bpmds08/program/paper1.pdf
100. Ma, Z., Wetzstein, B., Anicic, D., Heymans, S., Leymann, F.: Semantic business process repository. In: Proceedings of the Workshop on Semantic Business Process and Product Lifecycle Management (SBPM) in conjunction with the 3rd European Semantic Web Conference (ESWC), pp. 92–100 (2007). URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2007-20&engl=
101. Markovic, I., Costa Pereira, A., Francisco, D., Muñoz, H.: Procedings of workshops on service-oriented computing. chap. Querying in Business Process Modeling, pp. 234–245. Springer-Verlag (2009). DOI http://dx.doi.org/10.1007/978-3-540-93851-4_23
102. Markovic, I., Pereira, A.C.: Towards a formal framework for reuse in business process modeling. In: Proceedings of the 2007 international conference on Business process management, BPM'07, pp. 484–495. Springer-Verlag (2008). URL http://portal.acm.org/citation.cfm?id=1793714.1793769
103. Markovic, I., Pereira, A.C., Stojanovic, N.: A framework for querying in business process modelling. In: Procedings of Multikonferenz Wirtschaftsinformatik (MKWI 2008) (2008). URL http://ibis.in.tum.de/mkwi08/23_Semantic_Web_Technology_in_Business_Information_Systems/03_Markovic.pdf
104. Martin, D., Wutke, D., Leymann, F.: A novel approach to decentralized workflow enactment. In: Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC), pp. 127–136. IEEE Computer Society (2008). DOI http://dx.doi.org/10.1109/EDOC.2008.22
105. McClure, C.: Software Reuse: A Standards-Based Guide, 1 edn. IEEE Computer Society (2001)
106. McGregor, J.: Backtrack search algorithms and the maximal common subgraph problem. Software - Practice and Experience **12**, 23–34 (1982)

107. McIlroy, M.D.: Mass-produced software components. In: Proceedings of the 1st International Conference on Software Engineering, pp. 88–98 (1968). URL http://www.cs.dartmouth.edu/~doug/components.txt
108. Messmer, B.T., Bunke, H.: A new algorithm for error-tolerant subgraph isomorphism detection. IEEE Transactions on Pattern Analysis and Machine Intelligence **20**(5), 493–504 (1998). DOI http://dx.doi.org/10.1109/34.682179
109. Mietzner, R.: Using Variability Descriptors to Describe Customizable SaaS Application Templates. Technischer Bericht Informatik 2008/01, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany (2008). URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2008-01&engl=0
110. Mietzner, R., Ma, Z., Leymann, F.: An algorithm for the validation of executable completions of an abstract BPEL process. In: Proceedings of Multikonferenz Wirtschaftsinformatik (MKWI). GITO-Verlag, Berlin (2008). URL http://ibis.in.tum.de/mkwi08/29_XML4BPM-XML_Integration_and_Transformation_for_Business_Process_Management/02_Mietzner-XML4BPM-long.pdf
111. Mietzner, R., Unger, T., Leymann, F.: Cafe: A Generic Configurable Customizable Composite Cloud Application Framework. In: CoopIS 2009 (OTM 2009), *Lecture Notes in Computer Science*, vol. 5870, pp. 357–364. Springer-Verlag (2009). URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2009-75&engl=0
112. Mili, A., Chmiel, S.F., Gottumukkala, R., Zhang, L.: Managing software reuse economics: An integrated ROI-based model. Annals of Software Engineering **11**(1), 175–218 (2001). DOI http://dx.doi.org/10.1023/A:1012599304672
113. Mohagheghi, P., Conradi, R.: Quality, productivity and economic benefits of software reuse: a review of industrial studies. Empirical Software Engineering **12**(5), 471–516 (2007). DOI http://dx.doi.org/10.1007/s10664-007-9040-x
114. Morisio, M., Ezran, M., Tully, C.: Success and failure factors in software reuse. IEEE Transactions on Software Engineering **28**(4), 340–357 (2002). DOI http://dx.doi.org/10.1109/TSE.2002.995420
115. Petrov, I., Jablonski, S., Holze, M., Nemes, G., Schneider, M.: irm: An omg mof based repository system with querying capabilities. In: Conceptual Modeling - ER 2004, 23rd International Conference on Conceptual Modeling, Shanghai, China, November 2004, *Lecture Notes in Computer Science*, vol. 3288, pp. 850–851. Springer (2004)
116. Polyvyanyy, A., Kuropka, D.: A quantitative evaluation of the enhanced topic-based vector space model. Universitätsverlag Potsdam (2007). URL http://www.worldcat.org/isbn/3939469955
117. Polyvyanyy, A., Smirnov, S., Weske, M.: Process model abstraction: A slider approach. In: Proceedings of the 12th IEEE Enterprise Distributed Object Conference (EDOC). IEEE Computer Society (2008)

118. Poulin, J.S.: C.R.U.I.S.E. - Component Reuse in Software Engineering (2007)
119. Price, J.: Christopher alexanders pattern language. IEEE Transactions on Professional Communication **42**(2), 117–122 (1999). DOI http://dx.doi.org/10.1109/47.804820
120. Ramesh, R., Ramakrishnan, I.V.: Nonlinear pattern matching in trees. Journal of the ACM (JACM) **39**(2), 295–316 (1992). DOI http://doi.acm.org/10.1145/128749.128752
121. Raynard, B.: TOGAF The Open Group Architecture Framework 100 Success Secrets - 100 Most Asked Questions: The Missing TOGAF Guide on How to achieve and then sustain superior Enterprise Architecture execution. Emereo Pty Ltd (2008)
122. Rising, L.: Patterns: a way to reuse expertise. IEEE Communications Magazine **37**(4), 34–36 (1999). DOI http://dx.doi.org/10.1109/35.755446
123. Rosemann, M.: 22 potential pitfalls of process modeling. Business Process Management Journal **11&12** (2006). DOI http://dx.doi.org/10.1108/14637150610657567
124. Rosemann, M., van der Aalst, W.M.P.: A configurable reference modelling language. Journal of Information Systems **32**, 1–23 (2007). DOI 10.1016/j.is.2005.05.003
125. Russell, N., van der Aalst, W., ter Hofstede, A.: Exception handling patterns in process-aware information systems. Tech. rep., BPM Center (2006). URL http://www.workflowpatterns.com/documentation/documents/BPM-06-04.pdf
126. Russell, N., ter Hofstede, A., Edmond, D., van der Aalst, W.: Workflow resource patterns. Tech. rep., Eindhoven University of Technology (2004). URL http://www.workflowpatterns.com/documentation/documents/Resource%20Patterns%20BETA%20TR.pdf
127. Russell, N., ter Hofstede, A.H., Edmond, D., van der Aalst, W.M.: Workflow data patterns. Tech. rep., Queensland University of Technology (2004). URL http://www.workflowpatterns.com/documentation/documents/data_patterns%20BETA%20TR.pdf
128. Russell, N., Hofstede, A.H.M.T., van der Aalst, W.M., Mulyar, N.: Workflow control-flow patterns: A revised view. Tech. rep., BPM Center (2006). URL http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2006/BPM-06-22.pdf
129. Sakr, S., Awad, A.: A framework for querying graph-based business process models. In: Proceedings of the 19th international conference on World wide web, pp. 1297–1300. ACM (2010). DOI http://doi.acm.org/10.1145/1772690.1772906
130. Scheer, A.W., Nüttgens, M.: Aris architecture and reference models for business process management. In: Business Process Management, Models, Techniques, and Empirical Studies, pp. 376–389. Springer-Verlag (2000). URL http://portal.acm.org/citation.cfm?id=647778.734910
131. Schlieder, T., Meuss, H.: Querying and ranking xml documents. Journal of the American Society for Information Science and Technology **53**(6), 489–503 (2002). DOI http://dx.doi.org/10.1002/asi.10060

132. Schmelzer, H.J., Sesselmann, W.: Geschäftsprozessmanagement in der Praxis, 6 edn. Hanser Fachbuch (2007). (In German)
133. Schumm, D., Karastoyanova, D., Leymann, F., Strauch, S.: Fragmento: Advanced Process Fragment Library. In: Proceedings of the 19th International Conference on Information Systems Development (ISD'10), pp. 1–12. Springer-Verlag (2010). URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2010-52&engl=
134. Schumm, D., Leymann, F., Ma, Z., Scheibler, T., Strauch, S.: Integrating Compliance into Business Processes: Process Fragments as Reusable Compliance Controls. In: Proceedings of the Multikonferenz Wirtschaftsinformatik (MKWI'10), Göttingen, Germany, February 23-25, 2010, pp. 2125–2137. Universitätsverlag Göttingen (2010). URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2010-03&engl=
135. Schumm, D., Leymann, F., Streule, A.: Process Viewing Patterns. In: Proceedings of the 14th IEEE International EDOC Conference (EDOC 2010), pp. 89–98. IEEE Computer Society Press (2010). URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2010-50&engl=
136. Schumm, D., Turetken, O., Kokash, N., Elgammal, A., Leymann, F., Van Den Heuvel, W.J.: Business process compliance through reusable units of compliant processes. In: Proceedings of the 10th International Conference on Current Trends in Web Engineering, pp. 325–337. Springer-Verlag (2010). URL http://portal.acm.org/citation.cfm?id=1927229.1927263
137. Shasha, D., Wang, J.T.L., Shan, H., Zhang, K.: Atreegrep: Approximate searching in unordered trees. In: International Conference on Scientific and Statistical Database Management, p. 89. IEEE Computer Society (2002). DOI http://doi.ieeecomputersociety.org/10.1109/SSDM.2002.1029709
138. Simon, C., Dehnert, J.: From business process fragments to workflow definitions. In: Informationssysteme im E-Business und E-Government (EMISA 2004), pp. 95–106. EMISA (2004)
139. Soffer, P., Reinhartz-Berger, I., Sturm, A.: Facilitating reuse by specialization of reference models for business process design. In: Proceedings of 8th Workshop on Business Process Modeling, Development, and Support (BPMDS'07) (2005). URL lams.epfl.ch/conference/bpmds07/program/Soffer_5.pdf
140. Steinmetz, N., Toma, I. (eds.): Web Service Modeling Language (WSML) Final Draft (2008). URL http://www.wsmo.org/TR/d16/d16.1/.
141. Streule, A.: Abstract Views on BPEL Processes. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany (2009)
142. Tan, W., Fan, Y.: Dynamic workflow model fragmentation for distributed execution. Journal of Computer in Industry **58**(5), 381–391 (2007). DOI http://dx.doi.org/10.1016/j.compind.2006.07.004

143. Thom, L.H., Lau, J.M., Iochpe, C., Mendling, J.: Extending business process modeling tools with workflow pattern reuse. In: J. Cardoso, J. Cordeiro, J. Filipe (eds.) Proceedings of International Conference on Enterprise Information Systems (ICEIS), vol. EIS, pp. 447–452 (2007). URL http://dblp.uni-trier.de/db/conf/iceis/iceis2007-3.html#ThomLIM07
144. Thom, L.H., Reichert, M., Chiao, C.M., Iochpe, C., Hess, G.N.: Inventing less, reusing more, and adding intelligence to business process modeling. In: Proceedings of19th International Conference on Database and Expert Systems Applications (DEXA 2008), *Lecture Notes in Computer Science*, vol. 5181, pp. 837–850. Springer Verlag (2008). DOI 10.1007/978-3-540-85654-2_75
145. Thomas, L.T., Valluri, S.R., Karlapalem, K.: Margin: Maximal frequent subgraph mining. In: Proceedings of the Sixth International Conference on Data Mining (ICDM), pp. 1097–1101. IEEE Computer Society (2006). DOI http://dx.doi.org/10.1109/ICDM.2006.102
146. Trickovic, I.: Modularization and reuse in WS-BPEL. SAP Developer Network (2005). URL https://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/docs/library/uuid/7d26a4e1-0601-0010-b9a4-c815157e69af
147. Vanhatalo, J.: Process structure trees - decomposing a business process model into a hierarchy of single-entry-single-exit fragments. Ph.D. thesis, Institut für Architektur von Anwendungssystemen der Universität Stuttgart (2009)
148. Vanhatalo, J., Koehler, J., Leymann, F.: Repository for business processes and arbitrary associated metadata. In: Proceedings of the BPM Demo Session at the 4th International Conference on Business Process Management (BPM), vol. 203, pp. 25–31. CEUR Workshop Proceedings (2006). URL http://wi.wu-wien.ac.at/home/mendling/bpmdemo/paper4.pdf
149. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. In: Proceedings of the 6th International Conference on Business Process Management (BPM), vol. 5240, pp. 100–115. Springer Verlag (2008). DOI http://dx.doi.org/10.1007/978-3-540-85758-7_10
150. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and more focused control-flow analysis for business process models through sese decomposition. In: Proceedings of the 15th International Conference on Service-Oriented Computing (ICSOC), vol. 4749, pp. 43–55. Springer Verlag (2007). DOI http://dx.doi.org/10.1007/978-3-540-74974-5_4
151. Vanhatalo, J., Völzer, H., Leymann, F., Moser, S.: Automatic workflow graph refactoring and completion. In: Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC), vol. 5364, pp. 100–115. Springer Verlag (2008). DOI http://dx.doi.org/10.1007/978-3-540-89652-4_11
152. Wang, T.: Repo4BPEL, a business model repository. Master's thesis, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany (2009). Diplomarbeit No. 4712

153. Wang, Y., Maple, C.: A novel efficient algorithm for determining maximum common subgraphs. In: Proceedings of the Ninth International Conference on Information Visualisation, pp. 657–663. IEEE Computer Society (2005). DOI http://dx.doi.org/10.1109/IV.2005.11
154. Warnecke, H.J.: Die fraktale Fabrik - Revolution in der Unternehmenskultur. Springer Verlag, Heidelberg, (1995). (In German)
155. Weerawarana, S., Curbera, F., Leymann, F., Storey, T., Ferguson, D.F.: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More. Prentice Hall PTR (2005)
156. Weske, M.: Business Process Management - Concepts, Languages, Architectures. Springer Verlag (2007)
157. Wetzstein, B., Ma, Z., Filipowska, A., Kaczmarek, M., Bhiri, S., Losada, S., Lopez-Cobo, J.M., Cicurel, L.: Semantic business process management: A lifecycle based requirements analysis. In: Proceedings of the Workshop on Semantic Business Process and Product Lifecycle Management (SBPM) in conjunction with the 3rd European Semantic Web Conference (ESWC), vol. 251, pp. 1–11. CEUR Workshop Proceedings (2007). URL http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-251/
158. William, F., Giancarlo, S.: An empirical study of reuse, quality, and productivity. Tech. rep., Virginia Polytechnic Institute && State University (1997). URL http://eprints.cs.vt.edu/archive/00000473/
159. Witold, A., Konstanty, H., Monika, K., Raul, P., Dominik, Z.: NFP ontology for discovery and sharing Web services in distributed registries. In: Proceedings of the 22nd International Conference on Advanced Information Networking and Applications (AINAW) - Workshops, pp. 1416–1421. IEEE Computer Society (2008). DOI http://dx.doi.org/10.1109/WAINA.2008.267
160. Wutke, D.: Eine Infrastruktur für die dezentrale Ausführung von BPEL/Prozessen. Ph.D. thesis, Universität Stuttgart (2010). URL http://elib.uni-stuttgart.de/opus/volltexte/2010/5677. (In German)
161. Zachman, J.A.: A framework for information systems architecture. IBM System Journal **38**(2-3), 454–470 (1999). URL http://www.research.ibm.com/journal/sj/263/ibmsj2603E.pdf
162. Zhang, K., Statman, R., Shasha, D.: On the editing distance between unordered labeled trees. Information Processing Letters **42**(3), 133–139 (1992). DOI http://dx.doi.org/10.1016/0020-0190(92)90136-J
163. Zhao, X., Liu, C., Sadiq, W., Kowalkiewicz, M., Yongchareon, S.: WS-BPEL business process abstraction and concretisation. In: Proceeding of the 14th International Conference on Database Systems for Advanced Applications. Springer-Verlag (2009)

[All links are last followed on 19.11.2011]