Querying Monitoring and Integrating Web Applications

by

Anat Eyal

This work was carried under the supervision of

Prof. Tova Milo and Prof. Catriel Beeri

A Thesis submitted for the degree of Doctor of Philosophy

Submitted to the Senate of Tel Aviv University
August 2007
To my dear daughters Liron and Shiri

My source of inspiration
Acknowledgement

First and foremost, I would like to thank my advisors Prof. Catriel Beeri and Prof. Tova Milo for their remarkable dedication and excellent guidance throughout my studies for the PhD. Not only did they provide me with invaluable help during the problem definition and solution stages, but they also helped present and disseminate my work to the research community.

I am indebted to Tova Milo, who has been my mentor throughout this academic journey. Her knowledge, experience and bright ideas, along with her efficient approach and her constant support and encouragement, have enabled me to focus on the target and have thus greatly contributed to the quality of my study. I will always treasure the hours we have spent together. She has been my supervisor, my guide and my friend.

Catriel Beeri’s contribution to my work is invaluable. I have greatly benefited and enjoyed our meetings and joint work. He has provided important feedback on all aspects of this thesis, from the low-level technical details to the high-level aspects of the study as a whole. His clear and precise way of thinking, his ability to simplify complicated matters have certainly contributed to the research and to the clarity of the presentation of the results.

I would also like to thank Simon Kamenkovich and Alon Pilberg for their collaboration in some of the results of this thesis. Working with them has been a very enjoyable and rewarding experience.

Special thanks to Prof. Yossi Mathias who has given me the opportunity to join his research seminar and who has been most helpful in the process of submission of the thesis proposal.
During my stay at Tel Aviv University, I have enjoyed the company of my colleagues at the data base research group. Our interactions have ranged from in-depth technical discussions to fun activities, and they have been a source of joy and moral support.

I would also like to thank Ossnat Avihu, Vladimir Gamaley, Avi Telyas, Amnon Strasser for their assistance in the initial phases of this study.

I am grateful to Tova Kalish for her support and help in administrative matters.

Finally, my deepest thanks to my father and daughters for their loving support and continuous encouragement.
Abstract

As the number and variety of Web applications grow, more business opportunities emerge for providing new services based upon existing applications (e.g. virtual malls, comparative shopping). In particular, many enterprises nowadays use business processes for inter and cross-organizational applications. A business process (BP for short) consists of a group of business activities undertaken by one or more organizations in pursuit of some particular goal. It usually operates in a distributed environment and the software implementing it is fairly complex. Standards facilitate the design, deployment, and execution of BPs. In particular, the recent BPEL standard (Business Process Execution Language), provides an XML-based language to describe the interface between the participants in a process, as well as the full operational logic of the process and its execution flow. BPEL specifications are automatically compiled into executable code that implements the described BP and runs on a BPEL application server. Processes execution is traced, and their run-time behavior can be recorded in standard XML formats.

In this thesis we argue that these new standards not only simplify software development, but, more interestingly from an information management perspective, they also provide an important new mine of information. Queries about the BPs, that were extremely hard (if not impossible) to evaluate when the BP logic was coded in complex programs are now potentially much easier given a declarative specification. Furthermore, sophisticated querying, that interleaves static analysis of the BP specification with run-time process monitoring, can now be used for a variety of critical
tasks such as fraud detection, SLA (service level agreement) maintenance, and general business management. This provides an essential infrastructure for companies to optimize business processes, reduce operational costs, and ultimately increase competitiveness. To support these tasks, we present in this work two query languages: BP-QL and BP-Mon.

BP-QL (Business Processes Query Language) is a novel query language for querying business processes. It is based on an intuitive model of business processes, an abstraction of the BPEL standard and allows users to query business processes visually, in a manner very analogous to how such processes are typically specified. BP-QL can be employed in a distributed setting, where process components may be provided by distinct providers (peers). BP-Mon (Business Processes Monitoring), a novel query language for monitoring business processes, extends the BP-QL platform to monitor business processes executions using a query language. BP-Mon allows users to visually define monitoring tasks and associated reports, using a simple intuitive interface, similar to those used for designing BPEL processes. These languages have been implemented, and their performance and scalability were evaluated.

While the first two parts consider the querying and monitoring of existing BPs, the last part of this work investigates the construction of Web applications. We present the Application Manifold (AM), a framework for semi-automatic/system supported integration of existing, html-based Web applications. Based on data integration techniques used in semi-structured databases we simplify the process of integrating and composing Web applications.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgement</td>
<td>iii</td>
</tr>
<tr>
<td>Abstract</td>
<td>v</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>2</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>4</td>
</tr>
<tr>
<td>1.1.1 Recent History of Industrial Standards</td>
<td>4</td>
</tr>
<tr>
<td>1.1.2 Business Process Execution Language (BPEL)</td>
<td>5</td>
</tr>
<tr>
<td>1.2 New Challenges and Requirements</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Thesis Outline</td>
<td>7</td>
</tr>
<tr>
<td>2 Querying Business Processes with BP-QL</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>9</td>
</tr>
<tr>
<td>2.2 System Overview</td>
<td>14</td>
</tr>
<tr>
<td>2.2.1 Business Processes</td>
<td>15</td>
</tr>
<tr>
<td>2.2.2 The BP-QL Query Language</td>
<td>18</td>
</tr>
<tr>
<td>2.2.3 Query Semantics (informally)</td>
<td>22</td>
</tr>
<tr>
<td>2.3 The Formal Model</td>
<td>23</td>
</tr>
<tr>
<td>2.3.1 Simple Business Processes and Systems</td>
<td>24</td>
</tr>
<tr>
<td>2.3.2 Simple Queries</td>
<td>27</td>
</tr>
<tr>
<td>2.4 Issues in Answer Computation and Representation</td>
<td>30</td>
</tr>
<tr>
<td>2.4.1 On the Size of Results and Answers</td>
<td>30</td>
</tr>
<tr>
<td>2.4.2 The Key Ideas</td>
<td>32</td>
</tr>
<tr>
<td>2.5 The Answer Construction Algorithm</td>
<td>36</td>
</tr>
</tbody>
</table>
## 2.5.1 Preliminaries .......... 36
## 2.5.2 The Construction of $N[p^q \rightarrow p^s]$ and $R[p^q \rightarrow p^s]$ .......... 38
## 2.5.3 Construction of $N[[p^q \rightarrow p^s]]$ and $R[[p^q \rightarrow p_S]]$ .......... 41
## 2.5.4 A Comprehensive Example .......... 48

## 2.6 A Richer model .......... 53

## 2.7 Implementation .......... 61
## 2.7.1 Design Considerations .......... 61
## 2.7.2 Trade-offs .......... 67

## 2.8 Summary .......... 69

## 3 Monitoring Business Processes with BP-Mon .......... 71
## 3.1 Introduction .......... 71
## 3.2 Monitoring Business Processes .......... 75
## 3.2.1 Underlying Technology .......... 76
## 3.2.2 BP-Mon .......... 80

## 3.3 The Formal Model .......... 86

## 3.4 Matching and optimization .......... 92
## 3.4.1 Pattern Matching .......... 92
## 3.4.2 Optimization .......... 103

## 3.5 The full language .......... 106

## 3.6 Implementation and Experiments .......... 109
## 3.6.1 Implementation .......... 109
## 3.6.2 Experiments .......... 114

## 3.7 Summary .......... 118

## 4 Integration & Customization of Web Applications .......... 119
## 4.1 Introduction .......... 119
## 4.2 Preliminaries .......... 123
## 4.2.1 Data Model and Query Language .......... 125
## 4.2.2 Application Flow .......... 128
## 4.2.3 The Application Manifold Architecture .......... 129

## 4.3 Example .......... 131
CONTENTS

4.3.1 The Global Bookstore ........................................... 131
4.3.2 Local Bookstores ............................................... 134
4.4 How Things Work .................................................. 140
  4.4.1 Data ........................................................... 140
  4.4.2 Activities and Application Flow ................................. 148
  4.4.3 Putting It All Together ......................................... 150
4.5 Implementation ..................................................... 154
  4.5.1 The AM System ................................................ 155
  4.5.2 Specification and Code Generation ............................. 156
  4.5.3 Wrappers ....................................................... 161
  4.5.4 Run Time Operation ........................................... 163
4.6 A Posteriori Perspective to the AM ................................. 165
4.7 Summary ............................................................. 166

5 Related work ........................................................ 167
  5.1 The BP Model and Query Language ................................. 167
  5.2 Querying Data Streams ........................................... 170
  5.3 Process Integration ................................................. 173

6 Conclusion and Future Work ......................................... 177
  6.1 Querying BP Specifications ........................................ 178
  6.2 Monitoring BP Executions ......................................... 179
  6.3 Integrating BPs ..................................................... 180

Bibliography ............................................................. 181
Chapter 1

Introduction

As the number and variety of Web applications grow, more business opportunities emerge for providing new services based upon existing applications (e.g. virtual malls, comparative shopping). In particular, many enterprises nowadays use business processes for inter and cross-organizational applications.

A business process (BP for short) consists of a group of business activities undertaken by one or more organizations in pursuit of some particular goal. It usually depends upon various business functions for support, e.g. personnel, accounting, inventory; and interacts with other BPs/activities carried by the same or other organizations. Consequently, the software implementing such BPs is fairly complex, and typically operates in a cross-organization, distributed environment. *Standards* facilitate the design, deployment, and execution of BPs. It is common practice to use XML for data exchange between BPs, and *Web services* for interaction with remote processes [130].

In particular, the recent BPEL standard (Business Process Execution Language) [23], provides an XML-based language to describe the interface between the participants in a process, as well as the full operational logic of the process and its execution flow. BPEL specifications are automatically compiled into executable code that implements the described BP and runs on a BPEL application server. Processes execution is traced, and their run-time behavior can be recorded in standard XML formats.
This thesis addresses several challenges raised by the emerging Web technologies. We argue that these new standards not only simplify software development, but, more interestingly from an information management perspective, they also provide an important new mine of information. Queries about the BPs, that were extremely hard (if not impossible) to evaluate when the BP logic was coded in complex programs are now potentially much easier given a declarative specification. Furthermore, sophisticated querying, that interleaves static analysis of the BP specification with run-time process monitoring, can now be used for a variety of critical tasks such as fraud detection, SLA (service level agreement) maintenance, and general business management. This provides an essential infrastructure for companies to optimize business processes, reduce operational costs, and ultimately increase competitiveness. To support these tasks, we present in this work two query languages: BP-QL and BP-Mon.

BP-QL (Business Processes Query Language) is a novel query language for querying business processes. It is based on an intuitive model of business processes, an abstraction of the BPEL standard and allows users to query business processes visually, in a manner very analogous to how such processes are typically specified. BP-QL can be employed in a distributed setting, where process components may be provided by distinct providers (peers). BP-Mon (Business Processes Monitoring), a novel query language for monitoring business processes, extends the BP-QL platform to monitor business processes executions using a query language. BP-Mon allows users to visually define monitoring tasks and associated reports, using a simple intuitive interface, similar to those used for designing BPEL processes.

While the first two parts consider the querying and monitoring of existing BPs, the last part of this work investigates the construction of Web applications. We present the Application Manifold (AM), a framework for semi-automatic/system supported integration of existing, html-based Web applications. Based on data integration techniques used in semi-structured databases we simplify the process of integrating and composing Web applications.

To motivate our work, we first provide some background on BPs standardization efforts, and describe the BPEL standard. Next, we describe the challenges and requirements of querying, monitoring and integrating Web applications (Section 1.2).
CHAPTER 1. INTRODUCTION

Then we summarize our contributions towards addressing these new requirements in Section 1.3.

1.1 Background

1.1.1 Recent History of Industrial Standards

Web Services architecture is the emerging paradigm for implementing business processes within and across organizational boundaries. In this paradigm, the functionality provided by business applications is encapsulated within Web Services, which can be invoked by application programs or by other services through a stack of Internet standards including HTTP, XML, SOAP, WSDL [130], and UDDI [125]. These standards define Web Services discovery, description and activation, but do not provide mechanisms for describing how services can be connected to create business solutions.

In recent years many standards were proposed by different initiatives for a business process modeling language that will describe the business logic, i.e. how Web Services can interact with each other, the required messages, the execution order etc. Figure 1.1 (taken from [112]) shows the development of these specifications along time scale. Early work includes XLANG [134] which extends WSDL by control flow and provides support for sequential, parallel, and conditional process control flow, robust exception handling facility and long-running transactions. BPML [24] provides an abstracted execution model for collaborative and transactional business processes based on the concept of a transactional finite-state machine. BPSS [60] defines the collaborative process in terms of sequence of business transactions and message types and includes registry and repository standards for managing capabilities, security and QOS. WSFL [92] is a specification language based on Petri-nets for private and cross-organizational processes whose steps are implemented as Web Services and exposed in WSDL. WSCL, WSCL and WS-Choreography are standards for Web services choreography focused on modeling the sequencing of interaction between web services.
The recent BPEL standard (Business Process Execution Language [23], also identified as BPEL4WS), developed jointly by BEA Systems, IBM, and Microsoft, combines and replaces IBM’s Web Services Flow Language (WSFL) [92] and Microsoft’s XLANG [134]. Since the BPEL standard was submitted to OASIS, it seems to have gained the industry consensus support, this includes major players like IBM, Microsoft, Oracle, BEA Systems, SAP and others, and it is becoming the standard de-facto for BP specification and interaction. We next present the BPEL standard.

### 1.1.2 Business Process Execution Language (BPEL)

The BPEL standard is essentially a high level specification language with an XML-syntax that allows to describe a process’ execution flow and interaction with other processes. It provides an XML-based language to describe not only the interface between the participants in a process, but also the full operational logic of the process and its execution flow. A BPEL specification describes a process as a DAG consisting of activities (nodes) and links (edges) between them that detail the execution order of the activities. An activity is either atomic or compound. The atomic activities that can be used in a BPEL specification include operations such as invoke, for invoking an operation of some web service; receive, for waiting for a message from an external source; reply, for replying to an external source; assign, for copying data from one variable to another; throw, indicating errors in the execution; terminate, terminating the entire service instance; and empty, doing nothing. Compound activities are typically composed of several (atomic or compound) activities. Their types include sequence, where the component activities have sequential execution order; flow, where partial order is specified on component activities (possibly with parallelism), switch, for conditional execution; while, for looping; pick, for race conditions based on timing or external triggers; and scope, for grouping activities to be treated by the same
The data flow is actually handled by four types of activities: receive, reply, invoke and assign.

Because of the complexity of the BPEL XML-syntax, commercial vendors offer systems that allow users to design BPEL specifications via a visual interface, using a conceptual, intuitive view of the process, as a graph of activity nodes, connected by flow edges. Designs are automatically converted to BPEL specifications. These can be automatically compiled into executable code that implements the described BP and runs on a BPEL application server [112]. These servers allow to trace process instances, the activities they perform, messages sent or received by each activity, values of variables, etc. These events describe BPs run-time behavior, and can be recorded in standard XML formats.

1.2 New Challenges and Requirements

In this thesis we study how knowledge and techniques used in querying and integration of semi-structured databases can be employed to tackle the new challenges raised by the emerging Web technologies.

Querying Business Processes  Given that BPs are defined declaratively, they can be queried to learn about the processes and how they operate. The ability to answer such queries, in a possibly distributed environment, is of great practical potential for both individual users and organizations interested in cooperating, using, or analyzing BPs. This is extremely hard (if not impossible) when the BP logic is coded in a complex program. It is potentially much easier given a declarative specification like BPEL.

To support such queries, one needs an adequate query language, and an efficient execution engine for it. Firstly, it is important to be able to issue queries with high level of abstraction, as well as to zoom-in on all components. For this the query language should provide flexible granularity. Furthermore, BPs operate in a distributed environment, and specifications may reside on different servers. The
query language needs to support zooming into remote components, and also to query specifications on remote servers. We would also like to be able to retrieve parts of the flow. For example extract the path that answers the question - *What should I do to confirm my purchase?* And lastly, we would like to write queries in a graphical and intuitive way, in the same way we write the specifications.

**Monitoring BPEL executions**  So far we discussed the specifications, now we consider the executions. Monitoring the execution of BPs for interesting patterns is critical for enforcing business policies and meeting efficiency and reliability goals. Rather powerful tools (see discussion in Section 3.1) were developed for enterprise workflow management and address the needs of such companies. But the dynamic open nature of modern BPs pose new requirements, demanding, on the one hand, tighter surveillance, and on the other hand, a lighter, more ad hoc, deployment.

A monitoring tool should allow to define monitoring tasks in a high level of abstraction, using a visual and intuitive language. Monitoring tasks should be efficient, and should be easy to deploy. The language should support flexible description of execution patterns, i.e. sequential and parallel executions, and/or, repetitions. Reporting facilities should include the ability to issue notifications, create reports and allow invocation of corrective actions.

**Integrating and customizing Web applications**  Although many organizations adopt the Web service paradigm, there exists a large number of heterogenous http-based applications on the Web. This include various e-commerce applications like electronic shops, travel reservation systems, auctions and more. There is a need for convenient tools to support the integration and customization of existing applications.

### 1.3 Thesis Outline

To address these requirements, we present in this thesis two query languages: **BP-QL**, a novel graphical query language for querying Business Processes, and **BP-Mon** that extends the **BP-QL** platform to monitor business processes executions using a query language. In addition, we present the **Application Manifold** system which offers a novel
solution for specifying the integration and customization task of Web applications.

In Chapter 2, we describe $\text{BP-QL}$, a BP query language as well as its underlying formal model. We consider the properties of the various language components and explain how they influenced the language design. In particular we distinguish features that can be efficiently supported, and those that incur a prohibitively high cost, or cannot be computed at all. We also present our implementation, which complies with real life standards for business process specifications, XML, and Web services.

Next, in Chapter 3, we present $\text{BP-Mon}$, a BP monitoring language and its implementation, and describe our optimization techniques. An interesting feature of the implementation is that $\text{BP-Mon}$ queries are translated to BPEL processes that run on the same execution engine as the monitored processes. Our experiments indicate that this approach incurs very minimal overhead, hence is a practical and efficient approach to monitoring.

Finally, in Chapter 4, we consider the problem of integrating and customizing existing, http-based applications. The proposed framework supports a declarative specification language for specifying the integration and customization task, covering the full profile of the e-commerce applications. Then, acting as an application generator, the system generates a full integrated/customized e-commerce application. This part of the research took place before Web Services standards emerged. The results of this work are still relevant, and can easily be adapted to use the BPEL standard.

Related work is described in Chapter 5; we consider our results in perspective of state of the art in these areas. We conclude and discuss possible future directions in Chapter 6.

The materials in some chapters have been published as journal and conference papers. A first prototype of the query language presented in 2 was demonstrated in [16], a paper describing $\text{BP-QL}$ and the underlying model appeared in [17] and a comprehensive paper was submitted [18]. The prototype of the monitoring query language in Chapter 3 was demonstrated in [20], and a paper describing the query language was presented in [19]. The materials in Chapter 4 have been presented in [63] and a comprehensive paper was published in [64].


Chapter 2

Querying Business Processes with BP-QL

2.1 Introduction

A business process consists of a group of business activities undertaken by one or more organizations in pursuit of some particular goal. It usually depends upon various business functions for support, e.g. personnel, accounting, inventory, and interacts with other BPs/activities carried by the same or other organizations. Consequently, the software implementing such BPs typically operates in a cross-organization, distributed environment.

Declarative BPEL specifications greatly simplify the task of software development for BPs. More interestingly from an information management perspective, they also provide an important new mine of information. Consider for instance a user who tries to understand how a particular travel agency operates. She may want to find answers to questions such as: Can I get a price quote without giving first my credit card details? What should one do to confirm a purchase? What kind of credit services are used by the agency, directly or indirectly, (i.e. by the other processes it interacts with)? Obviously, such queries are of great interest to both individual users and to organizations interested in using or analyzing BPs. Answering them is extremely hard (if not impossible) when the BP logic is coded in a complex program. It is potentially
much easier given a *declarative specification* like BPEL. For an organization that has access to its own BPEL specifications, as well to those of cooperating organizations, the ability to answer such queries, in a possibly distributed environment, is of great practical potential.

To support such queries, one needs an adequate query language, and an efficient execution engine for it. To address this need, we present in this thesis BP-QL, a new query language which allows for an intuitive formulation of queries on BP specifications, and query execution in a distributed cross-organization environment.

Before presenting our language, let us highlight briefly some of the challenges in querying BP specifications in general, and BPEL ones in particular.

**Flexible granularity** BP specifications may be abstractly viewed as a set of *nested* graphs, possibly with *recursion*: The graphs structure captures the execution flow of the process components; The nesting comes from the fact that the operations/services used in a process are not necessarily atomic and may have a complex internal structure (which may itself be represented by a graph); The recursion is due to the fact that a process may call itself indirectly, through calls it makes to other processes. Users may wish to ask *coarse-grain* queries that consider certain process components as black boxes and allow for high level abstraction, as well as *fine-grained* queries that “zoom-in” on all the process components, possibly recursively. *An adequate query language must thus allow users to query the processes at different, flexible, granularity levels.*

**Distribution** As mentioned above, BPs typically operates in a cross-organization, distributed environment where each peer holds a set of BPs and may provide (resp. use) services to (of) remote peers. If a service’s internal flow has been defined in BPEL, and the service providers make this specification available to their cooperating organizations (say via a web service), *users may wish to zoom-in on these remote components as well to query the service specification.*

**Paths extraction** When querying BPs, users may be interested in retrieving, as answer, the qualifying *flow paths* (as for instance in the query “What should I do to
confirm my purchase?”). As the number of relevant paths may be large (or even infinite in recursive processes) a major challenge is to provide the users with a compact finite representation of the (possible infinite) answer.

**Ease of querying** As mentioned above, the BPEL standard offers an XML-based language for describing the operational logic of a BP. Since a BPEL specification is essentially an XML document, a natural question is why not query it directly, using XQuery? A key observation is that the BPEL XML format is (1) very complex and (2) was designed with ease of automatic code generation in mind; however, it is extremely inconvenient for querying. To express even a very simple inquiry about a process execution flow, one needs to write a fairly complex XQuery query that performs an excessive number of joins. Furthermore, even if a more query-friendly XML representation for it had been chosen (as indeed is done internally in our implementation), XQuery, as is, would still not be adequate for the task: XQuery only returns document elements, but not paths, it does not support querying at different levels of granularity, and it does not offer tools for controlling distributed querying. Last but not least, querying an XML representation is much more difficult than querying directly a conceptual model. Essentially, ease of querying requires an intuitive, conceptual, data model, coupled with a matching, equally intuitive, query language.

The BP-QL query language presented in this thesis addresses these issues. It is based on an intuitive model of BPs, an abstraction of the BPEL specification, along with a graphical user interface that allows for simple formulation of queries over this model. In a sense, it follows the same design principles that guided commercial vendors in the development of graphical editors for the specification of BPEL processes: it hides from the users the tedious BPEL XML details and allows for more natural query formulation. Indeed, we will see that the tight analogy between how BPs are specified in such editors and how they are graphically queried in BP-QL facilitate intuitive querying. BP-QL also offers facilities for controlling granularity and distribution in query formulation and allows paths in query results.

At the core of the BP-QL language are BP patterns that allow users to describe the pattern of activities/data flow that are of interest. BP patterns are similar to
the tree- and graph-patterns offered by existing query languages for XML [136] and graph-shaped data [46, 43, 114], but include two novel features designed to address the issues mentioned above. First, BP-QL supports navigation along two axis: (1) the standard path-based axis, that allows to navigate through, and query, paths in process graphs, and (2) a novel zoom-in axis, that allows to navigate (transitively) inside process components (local as well as remote ones) and query them at any depth of nesting. Second, paths are considered first class objects in BP-QL and can be retrieved, and represented compactly, even when involving activities performed on distinct peers.

Together, these features allow for simple formulation of queries on BPs. However, they make the evaluation of queries much more intricate than that of traditional XML/graph patterns. Indeed, some queries that can easily be evaluated on flat graphs/trees may become computationally expensive (or even undecidable) when nested graphs are concerned. To keep the evaluation of queries tractable, we had identified these problematic scenarios and carefully designed the language so that they are avoided, and polynomial-time query evaluation is guaranteed. Our analysis is based on modeling systems of processes and queries as graph grammars [62].

Observe that, in general, several modes of querying business processes are possible. One can query the specifications as data (e.g. “does the specification include a path from activity A to activity B”). One can also ask about patterns that may occur when the processes are executed (e.g. “can there be a run of the system where activity A is followed by activity B”). One can also monitor runs as processes execute, or pose queries on logs of past runs.

BP-QL is a query language for process specifications,¹ not about their possible runs. This is for two main reasons. First, querying the possible runs of a system is a verification problem [70] and is typically of very high complexity (from NP-hard for very simple specifications to undecidable in the general case [107]). Second, the analysis of runs requires a specification to have a well defined semantics. Unfortunately, BPEL is not based on a formal model [107]. To avoid these obstacles and guaranty complexity that is polynomial in the size of the data, BP-QL ignores the

¹A variant for monitoring and querying of executions is presented in the next chapter.
2.1. INTRODUCTION

run-time semantics of certain BPEL constructs such as conditional execution and variable values and focuses on the given specification flow. We believe this approach offers a reasonable balance between expressibility and complexity. Note that querying of specifications in fact “approximates” the querying of runs (e.g. only specifications that contain two given activities may potentially have runs where both occur). Hence, even when full run verification is desired, \textit{BP-QL} can be used as an efficient means to narrow the search space for the more costly, interpretation dependent, verification. It can also be used to select the process parts to be monitored at run time [122].

**Contributions** We now state the contributions of this work.

1. We present \textit{BP-QL}, a new graphical query language that allows for intuitive querying of process specifications, by offering a data model and an interface similar to those used for BPs \textit{specification}. It allows to retrieve paths, and offers facilities for querying at different levels of granularity, and for controlling distributed querying.

2. We present a formal model for systems of processes, and for our query language on such systems, based on \textit{graph grammars} [62]. This model allows to distinguish between query features that can be efficiently supported, and those that incur a prohibitively high cost, or cannot be computed at all. Using this model, we explain how to construct a finite and intuitive representation of the (possibly infinite) answers of queries in time polynomial in the size of the specifications.

3. Finally, we describe the system’s implementation, highlighting the main challenges faced and the solutions adopted.

This chapter is organized as follows. Section 2.2 introduces \textit{BP-QL} informally via a running example. The underlying formal model is presented in Section 2.3. We present the algorithm for construction of compact results in Section 2.5, and discuss some extensions to our model in Section 2.6. The system implementation is described in Section 2.7, and we summarize this chapter in 2.8;
2.2 System Overview

We present here an informal overview of BP-QL via a running example. To illustrate the features of BP-QL, we will consider a set of business processes (BPs) used by a consortium offering travel-related services. These include flight and hotel reservation, car rental, credit and accounting services. The processes, and their BPEL specifications, reside and operate on distinct peers. The specifications include the interactions between the various processes.

We first show how processes are specified, via the system’s graphical user interface, and then illustrate how they can be interrogated and queried with BP-QL. The graphical specification of BPs that we use is fairly standard, and is similar to those offered by commercial vendors (e.g. [112]). The novelty here is in the BP-QL graphical query language, designed especially for querying such specifications. The ease of query formulation is illustrated by comparing the query graphical interface to that used for the processes specification; there is a tight analogy between how processes are specified and how they are queried.

Running example Our running example is along the lines of W3C’s travel agent scenario[131]. Alpha-Tours, a fictional travel agency, offers to its potential clients the ability to book complete vacation packages: plane tickets, hotels, car rentals, and so on. The main steps of the reservation process are as follows: The user provides a destination, some dates, possibly some constraints, to the travel agency service. Next, the service obtains information about possible deals from airlines, hotels and car rental agencies and presents them to the user, which selects the ones she is interested in. Those are reserved by the agency. Finally, the user may cancel or confirm the reservation, passing her credit card details. The airline, car rental and hotel services contact a credit card service for payment authorization before they acknowledge the reservation.

We now demonstrate how the services are specified and queried. All screenshots were taken with our BP-QL visual designer and query tool.
2.2. SYSTEM OVERVIEW

2.2.1 Business Processes

A system consists of a set of BPs, possibly residing on distinct peers. A BP specification includes:

1. Some general description of the process properties, including its name, capabilities, the service provider, and so on.

2. The data used in the process, namely the process variables and the input and output parameters for the participating activities/services.

3. The activities of which the process is composed.

4. A description of the process operational and data flow.

Visually, the specification of a BP is represented as a directed labeled graph, with three types of nodes: property nodes (for 1), drawn as hexagons; data nodes (for 2), drawn as ellipses; and activity nodes (for 3), drawn as rectangles. Edges that connect data and activity nodes, called data flow edges, describe which data is read or output by which activity. Edges between activity nodes, called activity flow edges, describe the operational flow. To capture certain particular aspects of the operational flow of BPs, activity nodes may be identified as provided operations or requested operations. These describe the services offered by a process to other processes, and the external services that it requests, resp. Activity nodes may also be distinguished as atomic or compound. The latter represent invocations of composite (possibly remote) processes and are denoted by two little boxes at the top left corner of the activity icon. The interpretation of compound nodes is based on the ideas of statechart [77]: a zoom-in allows to replace a compound activity by a detailed description of the process that it invokes.

For illustration, consider the BP depicted in Figure 2.1. It represents the travel agency from our running example. The label under each node is its name. Each node carries some information on the process property/data/activity that it represents, which can be viewed by clicking on it. For instance, the property nodes at the top of the figure describe the process, its provider, and its capabilities. Most attributes of
these nodes are references to external taxonomies and ontologies that provide standard definitions of the service domain.\textsuperscript{2}

The process flow (on the left) and its data elements (on the right), are displayed in separate boxes. The small rectangles at the top and bottom of the activity flow are its entry and exit points. The BP contains four compound activity nodes, namely \texttt{searchTrip}, \texttt{reserveTrip}, \texttt{confirmTrip} and \texttt{cancelTrip}. A short thick incoming arrow indicates a \textit{provided operation}. A client may invoke each of the four provided operations (at the appropriate point in the flow). Edges between data nodes and activity nodes depict the data flow. For example, the client’s trip request is imported when the \texttt{searchTrip} activity is entered. The results are stored in a \texttt{tripResult} variable.

One can zoom into a compound activity node to see what is inside. Figure 2.2 shows the details of \texttt{searchTrip}. We can see that the travel agency interacts with other services to fulfill client requests. The short thick arrows outgoing of the

\textsuperscript{2}Implementation-wise they are stored in a UDDI repository.
2.2. SYSTEM OVERVIEW

Figure 2.2: Zooming into `searchTrip`.
searchCars, searchFlights, and searchRooms icons indicate that those are requested operations. Here, the node attributes (not displayed in the figure) provide the parameters (URL, operation name, ...) that allow one to invoke the relevant Web service. If the service providers make their BPEL specification available, one can zoom in also into these nodes as well to see the service specification.

The figure also shows data flow edges (for clarity some of these edges are omitted). For example, the set of airlines that the agency works with is imported when searchFlights is entered. The results of searching external airline services are stored in possibleFlights.

Before moving on to querying, we highlight two types of cycles that a specification may contain. First, the graph of a given BP may contain cycles, indicating that certain activities may be repeated an unbounded number of times. Second, in a system consisting of several processes that call each other, a BP may call itself indirectly, through calls it makes to other processes. This is another kind of cyclic structure: here one could zoom into the corresponding compound operation an unbounded number of times. Note that when querying BPs, users are often interested to retrieve flow paths as answers (as for instance in the query “What are the possible ways to purchase a plane ticket?”). In the presence of cycles, the number of qualifying paths may be infinite. One of the contributions of our work is to provide an intuitive, finite (and compact) representation for such possibly infinite answers.

### 2.2.2 The BP-QL Query Language

Given that BPs are defined declaratively, we can query the specifications to learn about the processes. In our running example, a user may want to ask questions like: 'Which services provide travel agency services to Europe, 'Which operations are provided by the travel agency service?', 'Which services are called directly or indirectly by the service?', 'Does the service allow to make a reservation without first giving credit card details? and if so, what does one need to do for making a reservation?'.

We proceed to explain how BP-QL can be used to express such queries.
2.2. SYSTEM OVERVIEW

Figure 2.3: Find services.

Figure 2.4: Find provided operations.

Figure 2.5: Credit services invoked when searching for trips.
BP-QL queries look much like the specifications. For querying BPs, BP-QL offers BP patterns which, intuitively, play for BPs a role analogous to that played for XML trees by tree pattern queries. They describe the pattern of activity/data flow that is of interest to the user and allow navigation along two axes: path-based and zoom-in based. Following the use of / and // in XPath[136] for denoting single and multiple step navigation, our PB patterns use edges with single and double heads to denote single and multiple edge paths, resp. Similarly, to allow a user to query about flows that are nested at any depth in the zoom-in hierarchy, compound activity nodes may have doubly bounded boxes, to denote an unbounded zoom in into the activities’ internal specifications. The nodes and edges of BP patterns can be associated with variables, and these can be used in selection conditions on their attributes and data and for joins. We also support negation (denoted by dashed nodes and edges).

We demonstrate the use of BP-QL via some example queries. Each query describes a process pattern that a user is looking for. The check boxes next to nodes and edges mark selected nodes and paths, resp., that the user wants to retrieve as the query result.

**Example 2.2.1** The query in Figure 2.3 searches for BPs that provide travel agency services and serves a certain geographical area, Europe. To compute such query, we first search the available UDDI registries (according to specific classifications) to get the list of services that satisfies our requirements. Then, we use this information to retrieve the service specifications and further computing of the query. The behavior box is the default box created automatically by the user interface. The user can draw the desired flow pattern by dragging operations to this box, as explained in the next example.

**Example 2.2.2** The query in Figure 2.4 searches for operations provided by the Alpha-Tours BP and the services it uses. The double headed edges inside the behavior box indicate that activities at any distance from the start/end nodes may qualify; the shape of the node restricts the search to provided operations. The double bounding of the behavior box denotes unbounded zoom-in; we look for operations provided by the BP and (transitively) the compound activities/services that it invokes. The zoom-in
2.2. SYSTEM OVERVIEW

Figure 2.6: (a) Negation (b) Path constraints.

is restricted to activities/services whose specifications reside on the same peer, since the deepSearch attribute is set to local. Setting it to global will extend the search to remote services as well.

Example 2.2.3 Figure 2.5 illustrates a join operation. The query checks which VISA credit card services are called (directly or indirectly) by the Alpha Tour’s confirmTrip activity. We use variables to define the join conditions. The join is value based, i.e. the nodes’ attributes are checked to have the same values.

Example 2.2.4 The query in Figure 2.6(a) illustrates the use of negation. It tests whether the users of Alpha Tours are never required to login when searching for flights. Formally, this is expressed by stating that a path to the searchFlights activity that passes through a login activity does not exists (dashed edges and nodes denote negation). The existing flow paths leading to searchFlights are then retrieved (as indicated by the small check box next to the double headed edge).

A more lenient query, that retrieves, the paths without a login leading to searchFlights, can be expressed by attaching a variable, say $x$, to the edge, along with the selection condition $x \in (\Sigma - \text{“login”})^*$. See Fig. 2.6(b). Regular path expressions as constraints on paths are discussed in Section 2.6.

Example 2.2.5 Finally, Figure 2.7 illustrates querying the data flow. The query searches for data elements that are (transitively) affected by the searchRequest, and serve as input for sending the suggested trips back to the client. By default, a double
headied edge between two data (resp. activity) nodes denotes paths consisting only of data (activity) flow edges. To override the default, (e.g. consider paths with all sorts of edges) one can attach, as above, a variable to the edge with an appropriate selection condition.

2.2.3 Query Semantics (informally)

When a query is evaluated, its patterns are matched against the system BPs. Its nodes and edges are assigned activity/data/property nodes and execution/data flow paths, resp. These are then used to construct the query result.

More precisely, the semantics of a query $q$ on system $S$ is defined as follows. An embedding is a function from the nodes and edges of $q$ to nodes, edges and paths of $S$, that satisfies the obvious constraints: Nodes are mapped to nodes of the same type, single/double-head edges are mapped to edges/paths between the corresponding end points. When a compound query node is doubly-bounded, nodes and edges in it may be mapped to nodes and paths in a process obtained by any number of zoom-ins into the activity's specification. For nodes and edges are associated with variables, the query constraints on these variables must be satisfied as well.

Each embedding defines one result for the query. The number of qualifying results may be large (possibly infinite in the presence of cycles). However, BP-QL provides a concise, intuitive (and finite) representation for the set. We illustrate this below with an example and provide more details on the construction in Section 2.3.
2.3 THE FORMAL MODEL

Example 2.2.6 Assume that the searchFlights service (invoked by searchTrip in Figure 2.2) has the structure depicted in Figure 2.8(a). The user can either login and check for the availability of various flights, or call, again, Alpha Tours’ searchTrip service to start a new search. Now, reconsider the query in Figure 2.6(b), that retrieves the paths leading to searchFlights that do not require a login.

Because of the potential cyclic service invocation, searchTrip can in fact be reached by an infinite number of paths, as depicted in Figure 2.8(b). Rather than listing all these paths, the user is provided with a compact representation (see Figure 2.9) that highlights the recursive structure of the results.

2.3 The Formal Model

In this section we briefly present the formal model underlying the BP-QL query language. We discuss the properties of the various language components and explain how these influenced our system’s design. In particular we distinguish features that can be efficiently supported, and those that incur a prohibitively high cost, or cannot be computed at all. To simplify the presentation we first consider a basic data model and query language, then enrich them to obtain the full fledged model.
2.3.1 Simple Business Processes and Systems

We assume the existence of two domains $\mathcal{N}$ of nodes and $\mathcal{L}$ of node labels. $\mathcal{L}$ is the disjoint union of several domains including data values, attribute names, data element names, process property names, and atomic and compound activity names. We assume some distinguished property names. These are introduced below, in the appropriate contexts.

Business Graphs and Processes

We model a (simple) BP as a directed labeled graph with nodes of two types: concrete and compound. Concrete nodes represent process properties, attributes, data elements, and atomic activities. Compound nodes represent compound activities, namely calls of (possibly remote) operations. Two distinguished nodes of the BP graph represent its start and end activities. Formally,

**Definition 2.3.1** A (simple) business graph is a pair $g = (G, \lambda)$, where $G = (N, E)$ is a directed graph in which $N \subseteq \mathcal{N}$ is a finite set of nodes, and $E$ is a set of edges with endpoints in $N$; and $\lambda : N \rightarrow \mathcal{L}$ is a labeling function for the nodes. Depending on their label type, we refer to the nodes in $g$ as activity nodes, value nodes, property nodes, etc. Nodes labeled by compound activity names are called compound nodes; all other nodes in $g$ are called concrete.
2.3. THE FORMAL MODEL

A (simple) business process (BP) is a triple \( p = (g, \text{start}, \text{end}) \), where: \( g \) is a business graph; \( \text{start} \), \( \text{end} \) are two distinguished activity nodes in \( g \); and each activity node in \( g \) resides on some path from \( \text{start} \) to \( \text{end} \).

Note that the start and end nodes need not be distinct. For example, a process may consists of just one activity node, which is both its start and its end. Also note that only activity nodes are restricted to be between the start and end nodes. Recall from section 2.2 that activities can be classified as requested or provided. This is modeled by assuming two particular property names provided and requested, and attaching to activity nodes appropriate property nodes.

For example, Figure 2.10 shows several BPs (ignore the “bubbles” for now). As before, we use squares for activity nodes and hexagons for property nodes. The leftmost BP has a single compound activity node, which is both its start and end. The one in the center has two distinct start and end nodes, and four provided operations. As mentioned above, compound nodes represent calls to composite operations. The internal structure of these operations is not part of the business process graph and is given separately, as we explain next.

Simple Systems

A system is a collection of business processes (or graphs), along with a mapping between compound nodes and their implementations – the processes they invoke. In the general case, a system may be distributed. This is ignored for now, for simplicity, and is discussed in Section 2.6.

Definition 2.3.2 A system \( S \) of business processes (resp. graphs) is a pair \( (P, \tau) \), where \( P \) is a finite set of business processes (graphs), and \( \tau \) is a (possibly partial) function, called the implementation function, from the compound activity nodes in \( P \) to business processes (graphs) in \( P \).

This definition can easily be extended to distinguish between root processes, that are directly accessible, and implementation processes, that are accessible only as

\[ ^{3} \text{In an actual system, } \tau(n) \text{ can be represented by attaching to } n \text{ the peer and process id (Web service URL, operation name, etc.) for the implementation of } n. \]
implementations of other processes. To simplify the presentation we omit this here.

The implementation function is partial when the internal structure of some compound activities is unknown (for instance when their providers do not wish to expose their specification). Recall from definition 2.3.1 that the only difference between business graphs and business processes is that the latter have distinguished start and end nodes. Systems of processes are used to model real life applications. Systems of graphs will prove useful to model query answers. For brevity, since we will mostly be dealing with systems of processes, unless stated otherwise the term system should be interpreted as system of processes.

Figure 2.10 shows a partial system. This is a partial description of the Travel Agency system from Figures 2.1, 2.2 (for simplicity, the data and attribute nodes are omitted). The full system should also contain, for example, the processes of the airline, car reservation, and hotel companies.

**System Refinement**

Given a system $S$, some BP $p$ in it, and a compound activity node $n$ in $p$, a more detailed description of $p$ (and hence of $S$) can be obtained by zooming-in and replacing the node $n$ by its implementation. We call this a refinement.

**Definition 2.3.3** Given a system $S = (P, \tau)$ and a BP $p$ in $P$, we say that $p \rightarrow p'$ (w.r.t. $\tau$) if $p'$ is obtained from $p$ by replacing some compound activity node $n$ in $p$ by its implementation $\tau(n)$. [Namely, $n$ is deleted from $p$, and a copy of the BP $\tau(n)$ is plugged in its place, with the incoming/outgoing edges of $n$ now being connected]
2.3. THE FORMAL MODEL

Figure 2.11: A refined system (after one step).

to the start/end nodes of $\tau(n)$, resp. If $n$ was the start/end node of $p$, the start/end node of $\tau(n)$ now takes this role.]

If $p \rightarrow p_1 \rightarrow \ldots \rightarrow p_k$ we say that $p_k$ is a refinement of $p$.

We say that $S \rightarrow S'$ (w.r.t. $\tau$) if $S'$ is obtained from $S$ by replacing the implementation $p$ of some compound activity node $n$ in $S$ by a refinement $p'$ of $p$. [Namely, a copy of $p'$ is added to $P$, the mapping $\tau$ for $n$ is updated to point to it, and $\tau$ is extended to map compound nodes in $p'$, to the same BPs as in $P$. Finally, if $p$ is no longer the implementation of any node, it is removed from $P$.]

If $S \rightarrow S_1 \rightarrow \ldots \rightarrow S_k$ we say that $S_k$ is a refinement of $S$. 

Note that if $S$ is a system, then each of its refinements is also a system. Figure 2.11 shows a refinement of the system from Figure 2.10, after one refinement step, in which the implementation of behavior was refined: the node labeled searchTrip has been "zoomed into" and replaced by its implementing process.

2.3.2 Simple Queries

We now consider queries and their answers. For simplicity we consider first simple positive queries without negation and joins. These, and other extensions, are
considered in Section 2.6.

Queries

Queries are modeled using BP patterns. These generalize BPs similarly to the way tree patterns generalize XML trees. The labels of nodes can be specified, or left open using *. Edges in a graph can be either single-headed, in which case they are interpreted over edges, or double-headed, in which case they are interpreted over paths. Similarly, nodes have a single or a double boundary, for searching only in the direct implementation of the node or in all its refinements, resp. We call edges with double head (resp. nodes with double boundary) transitive edges (nodes).

Definition 2.3.4 A BP pattern is a tuple \((p^*, T, R)\), where

1. \(p^*\) is a BP where nodes are labeled by elements from \(L \cup \{\ast\}\),

2. \(T\) is a distinguished set of edges and compound nodes in \(p^*\) called the transitive edges and nodes, resp.

3. \(R\) is a distinguished set of edges and nodes in \(p^*\) called the result edges and nodes, resp.

A simple query \(q\) is a system of BP patterns \((Q, \tau_q)\), where \(Q\) is a set of BP patterns, and \(\tau_q\) is an implementation function.

Query Answers

To evaluate a query, its patterns are matched to those of (refinements of) the system. A match is called an embedding.

Definition 2.3.5 Let \(q = (Q, \tau_q)\) be a simple query and let \(S\) be a simple system. An embedding of \(q\) into \(S\) is a homomorphism \(\rho\) from the nodes and edges in \(q\) to nodes edges and paths in some refinement \(S' = (P', \tau')\) of \(S\) s.t.

1. (component) each BP pattern in \(Q\) is mapped to one BP in \(P'\).
2. (nodes) each node is mapped to a node of the same type (property, value, activity, and so on); a node with a constant label is mapped to a node having the same label; each start (resp. end) node in \( q \) is mapped to a start (resp. end) node in \( S' \); and, each concrete[compound] activity node in \( q \) is mapped to a concrete[compound] activity node in \( S' \).

3. (edges) each (transitive) edge from node \( m \) to node \( n \) in \( q \) is mapped to an edge (path) from \( \rho(m) \) to \( \rho(n) \) in \( S' \).

4. (implementation) For each compound activity node \( n \) in \( q \), \( \rho \) maps the nodes and (transitive) edges in \( \tau_q(n) \) to nodes and edges (paths) in \( \tau'(\rho(n)) \). If \( n \) is not transitive then \( \tau'(\rho(n)) \) must be an original BP of \( S \) (i.e. not a refinement).

The result defined by \( \rho \) is the image under \( \rho \) of \( q \), restricted to its output nodes and edges. If the same node/edge occurs several times in the image, distinct copies are used for each occurrence.

The answer of \( q \), denoted \( q(S) \), is the set of all query results.

The conditions represent the intuition of what it means to map a pattern into a structure: The mapping needs to preserve both all information associated individual elements of the structure, and also its connectivity. Here, for nodes, their types and labels need to be preserved (nodes). As for connectivity, there are three levels here. First, connectivity of nodes by edges needs to be preserved (edges). Second, the partition of the structures into individual BPs needs to be preserved (component). Finally, the connectivity between BPs needs to be preserved (implementation).

Note that for activity nodes, the condition component is redundant: Since all activity nodes of a BP lie between the start and end nodes, the conditions nodes and edges imply the condition component for such nodes. However, as a query BP pattern may contain isolated nodes of other types, this condition is needed.
2.4 Issues in Answer Computation and Representation

The result associated with an embedding $\rho$ is, in general, a system of graphs. As there is no limit on the size of refinements, a result may be large, in terms of the sizes of the system and the query. Further, the number of results may be large, or even infinite. The latter may occur when the implementation function is recursive (i.e. cyclic), since then the number of refinements of a system may be infinite.

Thus, we face a double challenge: To compute answers efficiently, and to represent them in a compact manner.

We proceed to elaborate on the cases where results and answers may be large or infinite, then explain the main ideas of our solution. The answer construction algorithm is presented in Section 2.5.

2.4.1 On the Size of Results and Answers

We consider here the main factors that contribute to large or infinite results and answers, and provide some intuition about how they may be addressed. We consider first flat BPs, i.e. BPs with no compound activities, and then nested BPs, that is BPs with compound activities, and a non-trivial implementation function.

Flat BPs In this case, each query BP pattern may be considered in isolation. Still, even in this case, answers may be large or infinite. For example if the activity flow forks into several paths and then joins back, forks and joins again, and so on, several times, the number of possible paths is exponential in the number of forks. If the BP contains cycles, the number of paths that may match a given (transitive) query edge may be infinite.

The solution to this problem is easy: We can represent the set of paths between two nodes by a copy of the sub-graph that connects the nodes. One might actually say this is what the user intended: to see the specification of the paths between the two nodes, rather than the individual paths themselves. The details of this idea are
presented in Section 2.5.

**Nested BPs** Things become more complex in the presence of compound activities and a non-trivial implementation function. If each BP has several compound activities, then since these may be refined independently of each other, the number of refinements may be large. Additionally, a system may contain a recursive implementation function, hence have an infinite set of refinements, and there is no bound on the size of refinements. Since the results of a query are constructed from embeddings into all the refinements, results may be arbitrarily large, and there may be infinite number of results as well.

The solution here is based on viewing systems and queries as context free graph grammars [62], abbreviated CFGG. A CFGG is a finite set of graphs, where graphs may contain non-terminal symbols, and where grammar rules allow to replace a non-terminal by a graph from a given finite collection.

The intuition is that, for a system $S$, the implementation relationships correspond to grammar rules; the system refinements correspond to the graph language defined by the grammar. Similarly, a query $q$ can also be viewed as CFGG. The answer is obtained from the language of the query by homomorphisms (as defined in Definition 2.3.5) into the language of the system. We represent this language also as a CFGG, which can be efficiently constructed from the query and system CFGGs.

Note that this construction is related to “intersection” of the languages defined by the system and query grammars (followed by a “projection” that omits the portions that were not requested as output). In general, the intersection of two CFGG languages may not be a CFGG language [62]. (This generalizes the same property for string CFGs.) In our particular case, however, the query specification is sufficiently simple to guarantee the required closure: one can show that it belongs to a restricted class of CFGGs called recognizable sets [45], for which the intersection with another CFGG is known to yield a CFGG.

This implies that in principle we could try to adopt the intersection algorithm

---

4Confusingly, these are also called in the literature regular graph grammars. We will use only context free in this thesis.
presented in [45] to construct a finite representation for the query results. The problem, however, with this solution is that the algorithm of [45] is of high complexity - exponential in the size of the BPs\(^5\) - hence impractical for query evaluation. An important result of the present work was to detect that BP-QL queries form a subclass of the recognizable sets for which PTIME solution for our specific problem is possible, and to design such an algorithm.

In the rest of this section, and in the next one, we present and explain our algorithm, and the compact representation of answers. That is, we prove the following.

**Theorem 2.4.1** Query answers can be efficiently constructed and compactly represented. The construction time and the representation size are both polynomial in the size of the system \(S\) (with the exponent determined by the size of the query).

**Remark:** Note that in databases research one typically focuses on data complexity, as data is typically much larger than the query (e.g. complexity in relational queries is also polynomial in the size of data and exponential in the size of the query).

### 2.4.2 The Key Ideas

Our algorithm is based on a modular construction of a CFGG that describes the set of query results. It relies on the several ideas, explained below.

**Embedding and Result Decomposition**

The first idea is that each query result is a combination of smaller results that describe how one query process, say \(p^q\), is mapped to one system process, say \(p^S\).

Specifically, each embedding maps each \(p^q \in q\) into a process \(p^S \in S\), or to a refinement thereof (cf. the condition implementation). Let us call the restriction of an embedding to one query BP a BP embedding — a simple embedding if it maps the query BP to a system BP; a transitive embedding if it maps it to a refinement of a system BP.

\(^5\)Furthermore, to our knowledge, no PTIME algorithms for this intersection problem are known.
An embedding $\rho$ for the query can therefore be factored into a set of BP embeddings, one for each query BP, that together satisfy the condition (implementation).

We denote the set of all simple BP embeddings from $p^q$ to $p^S$ by $[p^q \rightarrow p^S]$, and the set of transitive BP embeddings by $[[p^q \rightarrow p^S]]$. Note that the BP embeddings in these two sets need to satisfy only the conditions (nodes) and (edges) of Definition 2.3.5.

Recall that each result is the image of the query under an embedding. It follows from the discussion above that a result can be decomposed into results for individual query BP’s. Each such result is called a BP result. It is a simple or a transitive result if it is obtained from a simple or a transitive embedding, respectively.

**Representation of BP Embeddings by Node Mappings**

The second idea is that many embeddings share the same underlying node mapping, and differ only on the assignments to the, possibly transitive, edges. Thus, the set of simple or transitive BP embeddings, from a query BP $p^q$ to a system BP $p^S$, can be “factorized” into groups of similar embeddings. The intuition is that these embeddings can be grouped based on their assignments to the nodes of $p^q$. Of course, when a node mapping from $p^q$ to $p^S$ is considered, one must ensure that it satisfies the conditions (nodes) and (edges).

Once an assignment $\rho$ for the nodes is chosen, the assignments for simple edges are determined, and for each transitive edge the set of paths that are the possible assignments is also determined. As suggested above, this set of possible path assignments for a transitive edge between two nodes is represented by a copy of the sub-graph that connects the nodes.

Formally, this intuition is expressed by the following.

**Proposition 2.4.2** For each embedding $\rho \in [p^q \rightarrow p^S]$, respectively $\rho \in [[p^q \rightarrow p^S]]$, every embedding $\rho'$ that is obtained from $\rho$ by replacing the image of some transitive edge $e \in p^q$ by some other path with the same endpoints, also belongs to $[p^q \rightarrow p^S]$, respectively $[[p^q \rightarrow p^S]]$.

The proposition holds since there are no constraints on the images of edges under embeddings, except that they connect the “right” nodes. As discussed in Section 2.6,
this proposition holds, with appropriate modifications, even when extensions such as
regular path expressions are added to the query language.

In Definition 2.3.5, the condition \(\text{(edges)}\) applies to embeddings. It can also apply
also to node mapping, with the following interpretation: A node mapping \(\rho\) satisfies
\(\text{(edges)}\) if for all nodes \(m\) and \(n\) in \(p^q\), and a (transitive) edge between them, \(\text{there exists}\) an edge (path) between \(\rho(m)\) and \(\rho(n)\).

**Corollary 2.4.3** For each node mapping \(\rho\) from \(p^q\) to \(p^S\), respectively to a refinement
of \(p^S\), that satisfies \(\text{(nodes)}\) and \(\text{(edges)}\), each of its extensions obtained by mapping
each simple, respectively transitive, edge from \(m\) to \(n\) to the edge, respectively a path,
from \(\rho(m)\) to \(\rho(n)\), is a member of \([p^q \rightarrow p^S]\), respectively \([|p^q \rightarrow p^S|]\). Conversely,
each member of \([p^q \rightarrow p^S]\), respectively \([|p^q \rightarrow p^S|]\), is obtained in this manner from
such a node mapping. ☑

We say that a node mapping \(\rho\) from \(p^q\) to \(p^S\), respectively a refinement of \(p^S\),
that satisfies \(\text{nodes}\) and \(\text{edges}\) represents the set of BP embeddings obtained from it
as described in the Corollary. By the Corollary, each such node mapping represents
a maximal subset of embeddings in \([p^q \rightarrow p^S]\), respectively \([|p^q \rightarrow p^S|]\), that agree on
nodes. The sets represented by different mappings are disjoint. A set of such node
mappings represents the union of the sets represented by its members.

Let us denote the set of node mappings from \(p^q\) to \(p^S\), respectively refinements of
\(p^S\), that satisfy \(\text{nodes}\) and \(\text{edges}\) by \(N[p^q \rightarrow p^S]\), respectively \(N[|p^q \rightarrow p^S|]\).

**Corollary 2.4.4** The sets of node mappings \(N[p^q \rightarrow p^S]\) and \(N[|p^q \rightarrow p^S|]\) represent
the sets \([p^q \rightarrow p^S]\) and \([|p^q \rightarrow p^S|]\), respectively. ☑

We note an important point about this representation. Assume that for each
pair \((p^q, p^S)\) of a query BP and system BP, a node mapping from \([p^q \rightarrow p^S]\) or from
\([|p^q \rightarrow p^S|]\) (as dictated by the query) is given. Select for each such node mapping one
of the embeddings it represents. If this collection of embeddings satisfies the condition
(implementation), then so does every other collection of embeddings obtained from
this set of node mappings. This follows from the fact that the implementation function
\(\tau\) is specified for nodes. That is, it suffices to check for the validity of this condition
for combinations of node mappings.
### Results representation

Further, since multiple BP embeddings that share the same assignment to nodes have a joint representation, so can the BP results obtained from them. Intuitively, if \( \rho \) is a node mapping, then the corresponding set of results consists of the images of the nodes under \( \rho \), and all edges or paths that connect them. We denote the set of results obtained from \([p^q \rightarrow p^S]\), respectively \([[[p^q \rightarrow p^S]]\), by \( R[p^q \rightarrow p^S] \), respectively \( R[[p^q \rightarrow p^S]] \). Then the observation is essentially that these two sets can be obtained from \( N[p^q \rightarrow p^S] \), respectively \( N[[p^q \rightarrow p^S]] \).

### Embeddings into Refinements

The third and last idea concerns the computation and representation of embeddings of a query BP pattern \( p^q \) into refinements of a system BP \( p^S \). Since \( p^S \) may have a large or even an infinite number of refinements, and refinements may be arbitrarily large, it is necessary to also decompose this task as well.

The important observation here is that an embedding from \( p^q \) to a refinement of the system process \( p^S \) consists of several parts. The first part maps some of the nodes and edges of \( p^q \) into \( p^S \) itself. Subsequent parts map additional nodes and edges of \( p^q \) into (refinements of) implementations of compound activity nodes of \( p^S \), and so on. The nodes and edges in the query pattern \( p^q \) that are not mapped to \( p^S \) itself, but into implementations of its compound nodes, must have appropriate structure. Specifically, the nodes and edges mapped into an implementation of a compound activity must form a sub-graph of \( p^q \), with a single entry and exit nodes.

Now, the same observation applies to the nodes and edges in these sub-graphs. If a sub-graph \( G \) is mapped to a refinement of an implementation \( \tilde{p}^S \) of a compound node of \( p^S \), then parts of \( G \) are mapped directly to \( \tilde{p}^S \), but certain sub-graphs are mapped into refinements of implementations of \( \tilde{p}^S \).

While for a specific mapping, this decomposition into sub-graphs may be arbitrarily deep, the involved sub-graphs are essentially all sub-graphs of \( p^q \).\(^6\) Thus, the set of mappings can be obtained by combining mappings of sub-graphs of \( p^q \) into system processes of \( S \). As explained in Section 2.5, this combination can be expressed as a

\(^{6}\)This is not really true in all cases, as explained in Section 2.5, but the intuition still holds for the construction explained there.
CHAPTER 2. QUERYING BUSINESS PROCESSES WITH BP-QL

Now, the graphs for which representations are constructed are (with a few exceptions) sub-graphs of the query patterns $p^q$. The construction terminates when mappings from each such graph to each system process have been considered, and combined into a CFGG. While the time required for this construction, and the space required for its outcome are exponential in the query size, they are polynomial in the system size.

2.5 The Answer Construction Algorithm

This section describes in detail how answers are represented, and the answer construction algorithm, following the ideas presented in the previous section.

Following the observation in Section 2.4.2, the construction of answer representations can be structured as follows: First, for each $p^q$ and $p^S$, construct the representations $N[p^q \rightarrow p^S]$ and $N[[p^q \rightarrow p^S]]$ for the two sets of BP embeddings $[p^q \rightarrow p^S]$ and $[[p^q \rightarrow p^S]]$. Then, construct images from these sets to represent $\mathcal{R}[[p^q \rightarrow p^S]]$, respectively $\mathcal{R}[[p^q \rightarrow p^S]]$. Finally, select combinations of elements from the sets that satisfy implementation.

Hence, we concentrate on embeddings from one query BP pattern $p^q$ to one system BP $p^S$. The algorithm for this construction is presented in several stages. The two main stages are the construction of $\mathcal{R}[[p^q \rightarrow p^S]]$ and $\mathcal{R}[[p^q \rightarrow p^S]]$. In each of these, we deal first with the easy case that there are no transitive edges, then consider the more difficult case where such edges are present.

2.5.1 Preliminaries

Context Free Graph Grammars

As explained in 2.3, a system $S$ can be viewed as a regular graph grammar[62]. We briefly discuss here these grammars.

An $m$-order context-free graph grammar (CFGG) is a quadruple $(N,T,P,I)$, where the components are as follows: $N$ is a set of nonterminal structures that are
polygons of size at most \( m \). That is they can be single nodes, pairs of nodes connected by an edge, triangles, and so on up to polygons with \( m \) vertices. \( T \) is a set of terminal elements. \( P \) is a finite set of rewriting rules of the form \( G \rightarrow H \), where \( G \) is a nonterminal structure and \( H \) a graph containing possibly both terminals and nonterminals. \( I \) is a set of initial graphs.

Rewriting of a graph using this CFGG corresponds to replacing a sub-graph that is isomorphic to some \( G \) that occurs in the left side of some rule by the graph \( H \) on the right side of the same rule. In this replacement, \( H \) needs to be connected to the rest of the graph in a manner that simulates the connections of \( G \). We skip the details, since we are interested only in a sub-class of this class of grammars. The language of the grammar is the set of graphs that can be obtained from \( I \) by repeated rewritings.

For our needs, 1-order grammars (denoted here by CFGG) are sufficient. In these, in each rule \( G \rightarrow H \), \( G \) is a single node (a compound activity node), and \( H \) is a graph with a single entry node, and a single exist node. When such a rule is used for a rewriting, each edge into \( G \) is pointed into \( \text{start}(H) \), and each edge from \( G \) becomes an edge from \( \text{end}(H) \).

It is easy to see that a system can be viewed as a CFGG. The implementation relationships correspond to grammar rules; the system refinements correspond to the graph language defined by the grammar. Similarly, a query \( q \) can also be viewed as a CFGG. The query answer consists of the graphs that are images of embeddings from the query BPs to system BPs.

The answer to the query will also be represented as a graph grammar. For conformance of notation with systems and queries, we will represent the set of productions as a multi-valued function \( \tau_A \) on node labels. For a label in its domain, this function associates a set of BP graphs. Thus, each node with this label can be refined (that is, replaced. ) In the discussion below, we don’t limit BP’s to have one root. This is just for convenience; to obtain a BP with a single root one can add a new root graph consisting of a single node, mapped by \( \tau_A \) to these multiple previous root graphs.
A Requirement from Results Representations

A basic requirement from a compact representation of a set of results is that one should be able to effectively obtain from it each individual result. A query is a pattern, that specifies a set of nodes and of related paths. Being able to obtain each path of each result is, by itself, insufficient. One needs to be able to identify each set that forms one result, that is a complete image of the query pattern.

To satisfy this requirement, we assume that the nodes of a query are uniquely identifiable (e.g. by integers, starting from 1). In result representation, nodes that are images of query nodes are associated with the id’s of the corresponding query nodes hence, in particular, they are marked as images of query nodes.

A BP graph with certain nodes marked as being derived from the query nodes is a representation of BP results from \( p^a \) to \( p^S \), respectively to a refinement of \( p^S \), if every BP graph obtained from it by selecting the marked nodes, adding edges between them for simple edges, and paths for transitive edges, is a simple (transitive) BP result from \( p^a \) to \( p^S \), respectively to a refinement of \( p^S \). A set \( R \) of BP graphs represents \( R[p^a \rightarrow p^S] \), respectively \( R[[p^a \rightarrow p^S]] \), if the union of the sets of results its members represent is equal to the corresponding set.

2.5.2 The Construction of \( N[p^a \rightarrow p^S] \) and \( R[p^a \rightarrow p^S] \)

In this construction, only simple embeddings from \( p^a \) to \( p^S \) are considered. We first present the case where no transitive edges are present in the query, then we consider the case where they are present.

Without transitive edges

Each node mapping that represents BP embeddings from \( p^a \) to \( p^S \) is essentially a homomorphism from \( p^a \) to \( p^S \) — this is the meaning of the condition (edges). The condition (nodes) filters out some homomorphisms. Finding a homomorphism from a graph \( G_1 \) to a graph \( G_2 \) is NP-complete in the size of \( G_1 \). Heuristics for constructing node mappings that satisfy (nodes) and (edges), that work well for simple graphs, are well known. We therefore assume that a representation \( N[p^a \rightarrow p^S] \) of \( [p^a \rightarrow p^S] \)
2.5. THE ANSWER CONSTRUCTION ALGORITHM

Figure 2.12: (a) Node mapping. (b) * label.

by node mappings has been constructed.

For this simple case, given a representation of BP embeddings as a node mapping $\rho$, the edges between nodes are uniquely determined. The only issue that needs to be addressed is that several nodes of $p^q$ may be mapped to the same node of $p^S$. In a result, there should be several copies of this node, one per query node mapped to it.

Thus, for each node mapping $\rho$, the representation of the corresponding set of results is an isomorphic copy of the query pattern, with each * label on a node $n$ replaced by the label on $\rho(n)$, and with each node associated with the id of the corresponding query node.

**Example 2.5.1** Figure 2.12(a) illustrates a BP $p^S$ comprised of nodes A, B, C with a loop on B and a BP pattern $p^q$. Then the mapping $\rho$ maps both B-nodes of $q$ to the same B-node of $p$. The answer contains a single result. The edge from $B_2$ to $B_3$ in the result is a copy of the loop edge from B to B in $p^S$. ✖

**Example 2.5.2** Let $p^S$ be as in the preceding example, and let $p^q$ be as described in Figure 2.12(b). Then there are two mappings, namely one that maps the node labeled * to the C labeled node, and another that maps it to the B-labeled. Thus, the answer now contains two results, as illustrated in the figure. ✖

**With transitive edges**

Two changes in the algorithm are needed here. First, by the condition (edges), if $e$ is a transitive edge that connects node $m$ to node $n$, then there must exist a path from
ρ(m) to ρ(n) in p. Hence, in the construction of node mappings, whenever p^q contains a transitive edge from m to n, when images for nodes m and n are considered, it must be checked that there is a path that connects them.

Second, since p^S may contain forks and joins or cycles, a transitive edge may be mapped to a large or an infinite number of paths. To address this issue, when a representation of results is constructed from a mapping ρ, if p^q contains a transitive edge from m to n, then all nodes and edges on all paths from ρ(m) to ρ(n) should be added to the representation. We add new copies of these nodes and edges to the representation of the result, that are not marked as images of query nodes. Hence, results can still be extracted unambiguously.

Example 2.5.3 Consider the example in Figure 2.13. There is only one node mapping from p^q to p^S, as illustrated in the figure. The representation of the answer contains a copy of a cycle in the system BP.

Proposition 2.5.4 The set N[p^q → p^S] of node mappings constructed as outlined above, is a representation of [p^q → p^S], the simple BP mappings from p^q to p^S.

The proposition follows directly from the construction, and from Corollary 2.4.4.

Corollary 2.5.5 The set of BP graphs, constructed as described above from the node mappings in N[p^q → p^S], is a representation of R[p^q → p^S].
2.5. THE ANSWER CONSTRUCTION ALGORITHM

2.5.3 Construction of \( N[p^g \to p^S] \) and \( R[p^g \to p_S] \)

So far, only simple embeddings from \( p^g \) to \( p^S \) were considered. Next, we consider transitive embeddings, that is embeddings into refinements of system processes.

In a transitive embedding, a new kind of cycle may exist: A part of \( p^g \) is mapped into \( p^S \), but some sub-graph of \( p^g \) may be mapped to an implementation \( p^S_1 \) of a composite action node of \( p^S \), the same or a smaller sub-graph may be mapped to an implementation \( p^S_2 \) of a composite action node in \( p^S_1 \), and so on, until a process is visited a second time, and possibly more times. Our approach here is to include each process on such a cycle only once, thus representing the possibly infinite set of paths compactly.

As in the case of simple embeddings, we first consider the case without transitive edges, and then generalize to include such edges.

No Transitive Edges

We first introduce some terminology for describing the parts of embeddings that are into (direct or indirect) implementations of compound action nodes of \( p^S \). Then we describe the construction.

In the construction below, new nodes with label \(*\) are generated. To be able to distinguish between the images of these nodes, and between them and images of the original query nodes, we assume a counter is maintained, whose initial value is \( n+1 \), where \( n \) is the number of nodes in the query. As a \(*\)-labeled node is generated, it is assigned as a unique id the value \( i \) of the counter, which is then incremented. In the discussion below, we refer to these nodes as \(*_i\) nodes.

Transitive sub-graphs Define a transitive sub-graph of a BP pattern \( g \) to be a non-empty sub-graph \( G \) of \( g \), such that all incoming edges arrive at a single node, called its entry and all outgoing edges depart from a single node, its exit. Clearly, only transitive sub-graphs of \( p^g \) may be mapped to implementations of compound action nodes of \( p^S \). Furthermore, if such a sub-graph is mapped to an implementation \( p \) as part of a transitive embedding, then its entry and exit are mapped to \( \text{start}(p) \) and
end(p), respectively.

We denote by trans(p) the set of transitive sub-graphs of a pattern p. Note that 
\( p \in \text{trans}(p) \) is possible. Also note that for \( g' \in \text{trans}(p) \), it holds that \( \text{trans}(g') \subseteq \text{trans}(p) \). For a transitive sub-graph \( g \), we denote by \( \hat{g} \) the pattern obtained from \( g \) by designating \( g \)'s entry, respectively exit, as start(\( g \)), respectively end(\( g \)). With these designations, in an embedding of \( g \) into a BP \( p \), the entry and exit nodes of \( g \) can only be mapped to start(\( p \)) and end(\( p \)), respectively.

In a given transitive embedding, several disjoint transitive sub-graphs may be mapped to implementations. A subset \( J \) of \( \text{trans}(p) \) that consists of disjoint sub-graphs is called a \( d \)-subset of \( \text{trans}(p) \). The set of all \( d \)-subsets of \( \text{trans}(g) \) will be denoted by \( \text{d-subs}(g) \). For a \( d \)-subset \( J = \{g_1, \ldots, g_k\} \) of \( \text{trans}(p) \), denote by \( p/J \) the pattern obtained from \( p \) by replacing the \( i \)'th element (\( i \leq k \)) of \( J \) in \( p \) by a new node labeled \( * \). For clarity, we denote this node as \( *_i \).

**The construction**

Given a transitive embedding \( \rho \) of \( p^a \) into \( p^S \), let \( J = \{g_1, \ldots, g_k\} \) be the subset of \( \text{trans}(p^a) \) of sub-graphs mapped by \( \rho \) into implementations of compound action nodes of \( p^S \). Then, \( \rho \) can be viewed as a combination of a simple embedding \( \hat{\rho} \) from \( p^a/J \) to \( p^S \), that maps each \( *_i \) to a compound action node \( f_i \) of \( p^S \), and additionally, transitive embeddings from each \( \hat{g}_i \) to \( \tau(f_i) \). These embeddings can be similarly decomposed.

For \( g \in \text{trans}(p^a) \), a (possibly empty) \( d \)-subset \( J \) of \( g \), and a BP \( p \) from \( S \), denote by \( < g/J \rightarrow p > \) the set of simple embeddings from \( \hat{g}/J \) to \( p \), such that each \( *_i \) is mapped to a compound action node of \( p \), and by \( \mathcal{R} < g/J \rightarrow p > \) the set of images of these embeddings. These sets can be constructed as follows: First, construct \( N[\hat{g}/J \rightarrow p] \). To obtain \( < g/J \rightarrow p > \), filter out the embeddings that do not satisfy the additional constraint that the images of \( *_i \) nodes are compound action nodes. From this, \( \mathcal{R} < g/J \rightarrow p > \) is obtained as described above, in Section 2.5.2.

For \( p^a \) itself, we use \( < p^a/J \rightarrow p^S > \) to denote the subset of \( [p^a/J \rightarrow p^S] \) of embeddings that satisfy the same constraint on \( *_i \) nodes, and \( \mathcal{R} < p^a/J \rightarrow p^S > \) for the derived results. The only difference is that we do not augment \( p^a \) with designated start and end node.
Now, the grammar that represents $\mathcal{R}[[p^q \to p^S]]$ is constructed as follows: Let $K_1 = \{ \mathcal{R} < p^q/J \to p^S > \mid J \in \text{d-sub}(p) \}$ and $K_2 = \{ \mathcal{R} < g/J \to p > \mid g \in \text{trans}(p^q), J \in \text{d-sub}(g), p \in S \}$. The graphs that are used in the grammar are $K_1 \cup K_2$.

As for the productions, assume that a graph was constructed from an embedding in which a node $^*i$ of $p^q/J$ or of $^*g/J$ was mapped to a compound node $f_i$ of $p$. Recall that when such a node is generated it is assigned a new id, say $h$, which also marks the image labeled by $f_i$. Let the $j$'th member of $J$ be $g_j$. Then, we set $\tau_A(f_h)$ to be the set $\{ \mathcal{R} < g_j/J_j \to \tau(f_i) > \mid J_j \in \text{d-sub}(g_j) \}$. Clearly, unreachable members can be removed from this grammar, by the usual method.

The set $\mathcal{R}[[p^q \to p^S]]$ is then the language defined by the grammar. Note that the symbols of the form $f_h$, that correspond to images of new $^*$-labeled nodes, are non-terminals. The language consists of graphs generated by the grammar that do not contain such non-terminals.

**Example 2.5.6** Figure 2.14 illustrates a system with two BPs, $p_1$ and $p_2$, in which $\tau(A) = p_2, \tau(B) = p_1$, and a BP pattern $p^q$. We want to construct $\mathcal{R}[[p^q \to p_1]]$.

We start by computing the set $K_1 = \{ \mathcal{R} < p^q/J \to p_1 > \mid J \in \text{d-sub}(p^q) \}$. For that, we need to consider the d-subsets of $p_q$. The transitive sub-graphs of $p^S$ are $g_1 = \{ ^*1 \}$, $g_2 = \{ A_2 \}$, $g_3 = \{ ^*3 \}$, $g_4 = \{ ^*1, A_2 \}$, $g_5 = \{ A_2, ^*3 \}$, $g_6 = \{ ^*1, A_2, ^*3 \} = p^S$. The d-subsets are then $J_0 = \emptyset$, the singleton set $J_i = \{ g_i \}$, and additionally $J_7 = \{ g_1, g_2 \}$, $J_8 = \{ g_1, g_3 \}$, $J_9 = \{ g_2, g_3 \}$, $J_{10} = \{ g_1, g_2, g_3 \}$, $J_{11} = \{ g_1, g_3 \}$, $J_{12} = \{ g_3, g_4 \}$. We illustrate the construction for some of these.
For \( J_0 = \emptyset \), we construct (two) simple solutions, illustrated in Figure 2.14.

For the other d-subsets, the major effort is to construct mappings for \( \hat{g}_i, 1 \leq i \leq 6 \). These can then be combined to produce results for all non-empty d-subsets.

For \( \hat{g}_1 \) and \( \hat{g}_3 \), direct mappings into \( p_2 \) do not exist, since in these two graphs, the start and end nodes are the same, but not so in \( p_2 \). For the other transitive sub-graphs, direct mappings also do not exist, since an \( A \)-labeled node must be mapped to an \( A \)-labeled node. Thus, the sets \( K_i = \{ \mathcal{R} < g_i/J \rightarrow p_2 > \mid J \in \text{d-sub}(g_i), 1 \leq i \leq 6 \} \), if they are non-empty, contain only members derived from non-empty \( J \)s.

For \( \hat{g}_i, 1 \leq i \leq 3 \), there is just one choice of a non-empty d-subset, namely the graph \( \hat{g}_i \) itself. The graph \( \hat{g}_i \) needs then to be mapped to \( p_1 \). No mapping exists, for the same reason that no mapping to \( p_2 \) exists. It is easy to see that, for similar reasons, there are no mappings from \( \hat{g}_i, 4 \leq i \leq 6 \) into \( p_1 \).

It remains to consider proper transitive sub-graphs of \( \hat{g}_i, 4 \leq i \leq 6 \). We leave it to the reader to show that no mappings can be found here as well. Thus, in summary, only the mapping for \( J_0 \) produces a result.

With transitive edges

The new issue to be dealt with here concerns the possibility that a transitive edge is mapped to a path that is both in a process \( p \), and in some implementation of a compound activity node of \( p \).

Assume the pattern \( g \) contains a transitive edge from \( m \) to \( n \). In constructing transitive embeddings from \( g \) to \( p \), the following options should be considered. In the discussion below, we use \( p' \) as a generic name for some implementation of a compound activity node of \( p \).

- The path corresponding to the transitive edge from \( m \) to \( n \) is fully in \( p' \). In this case, there is a transitive sub-graph \( G \) that contains the edge. This includes the case that \( m \) is the entry of \( G \), or \( n \) is its exit, or both. In this case, the edge will be dealt with when embeddings from \( G \) to \( p' \) are considered.

- The transitive edge from \( m \) to \( n \) is mapped to a path that starts in some \( p' \), and ends in \( p \). In particular, the image of \( m \) is in \( p' \) and that of \( n \) is in \( p \). To
2.5. THE ANSWER CONSTRUCTION ALGORITHM

Figure 2.15: With transitive edges.

construct a suitable transitive sub-graph \( G \), we need to add a new node to serve as the entry of \( G \). We decompose the edge by adding a new node with a * label, and replacing the edge with two transitive edges, one from \( m \) to * and the other from * to \( n \). Any graph whose entry is this new node, and also contains \( n \) is now also considered as a transitive sub-graph.

- The dual case that the graph contains \( n \) but not \( m \) is treated similarly.

- For the case that the path starts in \( p \), contains \( m \), proceeds to some \( p' \) and returns to \( p \), where it contains \( n \) we add two new * nodes, to serve as the entry and exit of a transitive sub-graph \( G \).

With the above modifications in place, d-subsets of transitive sub-graphs can be constructed, and the algorithm outlined in Section 2.5.2 above, modified to deal with transitive edges, as in Section 2.5.2 can be used. One additional point to be considered is that the newly introduced * nodes just described were not present in the original query. Their images, therefore, are not marked as query node images. Let us denote the grammar constructed as above by \( \mathcal{G}[p^q / \rightarrow p^s] \).

**Example 2.5.7** Consider the system and query in Figure 2.15. \( S \) consists of three BP’s, namely \( p^s_1, p^s_2, p^s_3 \). The query \( q \) consists of \( p^q_1, p^q_2 \). Then \( A_1 \) in \( p^q_1 \) is mapped to \( A \) in \( p^s_1 \). We need now to find results derived from embeddings of \( p^q_2 \) into a refinement of \( p^s_2 \).

Let us introduce a new * node into the double headed edge from \( D_2 \) to \( C_3 \) (see Figure 2.16 on the left), then replace the path from \( D_2 \) to *1 by *5 (see Figure 2.16 on
CHAPTER 2. QUERYING BUSINESS PROCESSES WITH BP-QL

Figure 2.16: Graph modifications.

Figure 2.17: Answer.

the right). Now, the path from \( \ast_5 \) to \( C_3 \) can be mapped to \( p_2^q \), giving \( B_5 \rightarrow C_3 \), where \( B_5 \) is a non-terminal of the grammar. As for the graph that includes a transitive edge from \( D_2 \) to \( \ast_4 \), with \( D_2 \) marked as the start node and \( \ast_4 \) marked as the end node, it can be mapped to \( P_3^q \), giving \( D_2 \rightarrow E \). Now we set \( \tau(B_5) \) to this graph as depicted in the answer in Figure 2.17. Note that the image of \( \ast_4 \) is not marked as a query node image, nor as a non-terminal.

This is the desired result, derived from an embedding of \( p_2^q \) into a refinement of \( p_2^s \).

Proposition 2.5.8 The language of \( \mathcal{G}[[p^q/\rightarrow p^s>]] \) is a representation of the set of results \( \mathcal{R}[[p^q \rightarrow p^s]] \). Specifically, each graph in the language represents a set of members of \( \mathcal{R}[[p^q \rightarrow p^s]] \) that are derived from a shared node mapping. Conversely, each set of results derived from a shared node mapping is represented by a graph in the language of \( \mathcal{G}[[p^q/\rightarrow p^s>]] \).

Proof: (outline)
\( \Rightarrow \): This direction is proved using induction on derivation depth. A derivation of
2.5. THE ANSWER CONSTRUCTION ALGORITHM

a graph in the language starts from \( R < p^\theta / J \rightarrow p^S > \), for some \( J \) in \( d\text{-subs}(p^\theta) \). This gives, by Proposition 2.5.4, a result derived from a node mapping of \( p^\theta / J \rightarrow p^S \), that maps every new \( * \) node to a compound activity node \( f \). The graph is obtained by replacing each such \( f \) by the language of the same grammar, with roots \( \{ R < g/K \rightarrow \tau(p) > \} \), where \( g \) ranges over \( J \), and \( K \) is a d-subset of \( \hat{g} \). By induction on derivation depth, these represent results obtained from transitive embeddings of \( \hat{g} \) into \( \tau(f) \). The claim follows.

\( \Leftarrow \): This direction works by induction on the refinement depth. Let us first define this concept.

Assume \( p' \) is a refinement of \( p \). The refinement depth of \( p' \) with respect to \( p \) is defined as follows. The refinement depth of \( p \) with respect to itself is zero. If \( p \neq p' \), then \( p' \) is obtained by replacing some compound nodes of \( p \) by implementations. If the maximum refinement depth of these implementations with respect to their original processes is \( m \), then the refinement depth of \( p' \) with respect to \( p \) is \( m + 1 \).

Assume given a set of results derived from a (maximal) set of transitive embeddings from \( p^\theta \) to \( p'^S \), where \( p'^S \) is a refinement of \( p^S \) of depth \( m \), that share a node mapping \( \rho \). The part of \( \rho \) in \( p^S \) defines a d-subset \( J \). The derivation of the grammar starts from \( R < p^\theta / J \rightarrow p^S > \). If \( J = \emptyset \), then \( p'^S = p^S \) and we are done. Otherwise, each remaining part, say \( \rho_i \) of the node mapping, that corresponds to a \( G_i \) member of \( J \), is to a refinement \( p'^S_i \) of \( p^S_i \), where \( p^S_i \) is the implementation of a compound activity node of \( p^S \). The refinement depth of \( p'^S_i \) with respect to \( p^S_i \) is smaller than \( m \). Hence, the set of results derived from \( \rho_i \) can be represented by the grammar, with the roots replaced by \( \{ R < \hat{G}_i/p^S_i \rightarrow K > | K \in d\text{-subs}(\hat{G}_i) \} \). From this, the claim follows.

\( \Box \)

Putting It All Together

To summarize, we now have representations for results obtained from simple and transitive embeddings, from each \( p^\theta \) to each \( p^S \). Note that the representations for simple embeddings may be viewed as degenerate grammars.

Now, consider the combinations of representations, one for each combination of \( p^\theta \in Q \) and \( p^S \in S \). We discard those that do not satisfy (implementation). In particular, if a representation is obtained from a node mapping \( \rho \) from \( f \) to \( p \) that
maps a transitive query node \( n \) to a compound activity node in \( p \), then \( \tau_q(n) \) must be mapped transitively to \( \tau(\rho(n)) \).

**Theorem 2.5.9** The construction outlines above generates a finite representation of the set of results for \( Q \) on \( S \).

A more efficient version of the algorithm works incrementally. We first construct representations for embeddings from members of \( Q_r \) to members of \( S_r \). Then, we construct representations for new pairs on demand, to satisfy implementation. Details are left to the reader.

To summarize, we are now in position to consider Theorem 2.4.1, that the construction time and the representation size are both polynomial in the size of the system. Indeed, in the construction above, in the worst case two representations (one regular, one transitive) are constructed for each combination of a transitive sub-graph of a query process \( p^q \) and a system process \( p^S \). The number of transitive sub-graphs of the query processes may be exponential in the query size. Each construction may take time that is exponential in the transitive sub-graph size.

### 2.5.4 A Comprehensive Example

We now present a comprehensive example to demonstrate the algorithm described above. Consider the system described in Figures 2.1 and 2.2, and the refinement of `searchFlights` described in Figure 2.8(a). Note that although Figures 2.1 and 2.2 illustrate the travel agency with a refinement of some activities, the system is composed of four processes: the behavior, the Alpha-Tours’ main process which implements it, the implementation of the `searchTrip` activity, and the external `SearchFlights` process. Also reconsider the query described in Figure 2.6(b), that retrieves the paths leading to `searchFlights` that do not require a login. Recall that because of the potential cyclic service invocation, `searchTrip` can in fact be reached by an infinite number of paths, as depicted in Figure 2.8(b). In the following example we demonstrate the steps of the answer construction algorithm that computes a concise answer. Rather than listing all these paths, the answer represents the recursive structure of the results as described in Figure 2.9.
2.5. THE ANSWER CONSTRUCTION ALGORITHM

Figure 2.18: System.

Figure 2.19: Query.
Example 2.5.10 Figure 2.18 illustrates the formal representation of the system. The formal representation of the query is illustrated in Figure 2.19. We have here a system $S = (P, \tau)$ and BPs, $P = \{p_1^S, p_2^S, p_3^S, p_4^S\}$, $\tau(behavior_{p_1}) = p_2$, $\tau(searchTrip_{p_2}) = p_3$, $\tau(searchFlights_{p_3}) = p_4$, $\tau(searchTrip_{p_4}) = p_2$; and a query $q = (Q, \tau_q)$ with BP patterns $Q = \{p_1^q, p_2^q\}$. The query includes a transitive node (in $p_1^q$) and transitive edges (in $p_2^q$). Note that as mentioned in Subsection 2.3.1, the start and end nodes need not be distinct; $p_1^S$ consists of just one activity node, which is both its start and its end.

First, for each $p^p$ and $p^S$, we try to construct the representations $N[p^p \rightarrow p^S]$. We map the node in $p_1^q$ to $behavior$ in $p_1^S$, and the nodes in $p_2^q$ to the nodes in $p_3^S$. However, this mapping satisfies (nodes) and (edges) but not (implementation). We continue to the next step of constructing $N[[p^p \rightarrow p^S]]$.

As the first step of $N[[p^p \rightarrow p^S]]$ and $\mathcal{R}[[p^p \rightarrow p_S]]$, we consider transitive
embeddings, that is embeddings into refinements of system processes. The set of transitive sub-graphs $trans(p^q)$ of $p^q_2$ includes the following: $g_1 = \{\text{start}\}, g_2 = \{\text{searchFlights}\}, g_3 = \{\text{end}\}, g_4 = \{\text{start, searchFlights}\}, g_5 = \{\text{searchFlights, end}\}, g_6 = \{\text{start, searchFlights, end}\}$. From here we construct $[p^q \rightarrow p^S]$, according to the algorithm. We omit here some of the steps, and continue only with $g_2$. We now replace the $\text{searchFlights}$ node by a new node denoted $*_5$, that maps to an implementation that includes $\text{searchFlights}$ augmented with start and end nodes, as depicted in Figure 2.20.

Next, we reconstruct nodes mappings. In this step $p^q$ includes $p^q_1, p^q_2, p^q_3$. We map $*_5$ to a compound node in $p^S_2$. Since $S$ is partial, and includes only the implementation of $\text{searchTrip}$, we map $*_5$ to $\text{searchTrip}$. We also map $\text{searchFlights}$ in $p^q_3$ to $\text{searchFlights}$ in $p^S_3$. From this step we derive the first result by adding edges mappings. Since here we have transitive edges, we add all paths between the mapped nodes to the result representation, as depicted Figure 2.21 (on the top).

We continue with constructing transitive embeddings. After mapping $\text{searchFlights}$, we also need to consider transitive edges. To handle the possibility that a transitive edge is mapped to a path that is both in a process $p^S_3$, and in some implementation of a compound activity node of $p^S_3$, we replace each transitive edge in $p^q_3$ by a new node with two transitive edges. We add the new nodes $*_8$ and $*_9$ as depicted in Figure 2.20.

The set of transitive sub-graphs of $p^q_3$ includes the following nodes: $g_1 = \{\text{start}\}, g_2 = \{*_8\}, g_3 = \{\text{searchFlights}\}, g_4 = \{*_9\}, g_5 = \{\text{end}\}, g_6 = \{\text{start, *}_8\}, g_7 = \{*_8, \text{searchFlights}\}, g_8 = \{\text{searchFlights, *}_9\}, g_9 = \{*_9, \text{end}\}, g_{10} = \{\text{start, *}_8, \text{searchFlights}\}, g_{11} = \{*_8, \text{searchFlights, *}_9\}, \ldots$.

From here we construct $[[p^q \rightarrow p^S]]$, according to the algorithm. We omit here some of the steps, and continue only with $g_{11}$. We replace $\{*_8, \text{searchFlights, *}_9\}$ with the new node $*_{10}$, that maps to an implementation that includes $\text{searchFlights}$, and $*_8, *_9$ denoted as the start node and end node respectively (see the third step in Figure 2.20). We reconstruct node mappings. We try to map $p^q_4$ to $p^S_4$, we map $*_{10}$ to $\text{searchTrip}$ in $p^S_4$, and map its implantation recursively to $p^S_2$. Then by constructing transitive embedding (following recursion) we derive the second answer, as depicted
Figure 2.21: Answer.
2.6  A Richer model

The query model discussed in Sections 2.3 and 2.5 was rather simple. This simplicity allowed us to concentrate in Section 2.5 on the main ideas of how answers of queries are represented. We now present some useful extensions that enhance the expressive power of the language, and facilitate the querying of real life business processes. Some of these were illustrated in Section 2.2.

Negation  In a query with negation, the patterns have some nodes and edges that are distinguished as negative. The intuitive interpretation is that the query searches for occurrences of the positive portions of the patterns, for which none of the negative parts co-occur.

More formally, to define the semantics of queries with negation we extend the notion of embedding:

**Definition 2.6.1**  Let \( q = (Q, \tau) \) be a query with negation (i.e. a query whose graph patterns have some nodes and edges that are distinguished as negative), and let \( S \) be a simple system. We denote as positive\((q)\), the positive part of \( q \), obtained from \( q \) by deleting all the negative edges and nodes, and all the edges incident on these nodes.

An embedding of \( q \) into \( S \) is a homomorphism \( \rho \) from the nodes and edges in positive\((q)\) to nodes edges and paths in some refinement \( S' = (P', \tau') \) of \( S \) s.t.

1. The conditions nodes, edges and implementation of Definition 2.3.5 are satisfied.

2. \( \rho \) cannot be extended to an embedding of any query \( q' \) obtained from positive\((q)\) by adding all the negative nodes and edges of some of its BP patterns.

A finite representation for the query answer can be constructed essentially as explained in Section 2.5. Note that both embeddings for the positive part and for the
full query can be grouped by shared node mappings. In the construction, each node mapping for the positive part that can be extended to a node mapping for the full query is eliminated.

**Label predicates and regular path expressions** The simple queries considered so far only allow nodes with a particular label or *. But sometimes one may be interested in system nodes that conform to certain conditions. For instance, rather than searching for the `searchFlights` activity, we may want to retrieve all the activities whose name contains the string “search”. This can be achieved by using *label predicates*. In an embedding, a query node labeled by a label predicate must be mapped to system node whose label satisfies the predicate. This can easily be accommodated in our answer construction algorithm.

Another useful feature is regular path expressions on node labels. Transitive edges in the query may be annotated by regular expressions. In an embedding, such edges must be mapped to paths such that their label sequence forms a word in the corresponding regular language.

**Example 2.6.2** Let us reconsider Example 2.5.3: Figure 2.22(a) illustrates the modified example. The edge from $B_2$ to $D_3$ in $p^a$ is labeled with the regular expression $(CB)^2C$. Without this expression, the answer was as illustrated at 2.13, and includes a cycle. With it, it is converted to the answer in Figure 2.22.

The construction of a finite representation for the query answer extends naturally
to support this extension. First, replace the transitive edge by an NFA that represents the regular expression with the edge labels pushed forward to become node labels (in this pushing, epsilon labels are simply absorbed). Then, apply the answer construction algorithm. Note that the nodes of the newly introduced NFA are not considered as query nodes.

**Example 2.6.3** Consider the same system BP and query BP as in Example 2.6.2, except that the regular expression is \((CB)^*C\) (see Figure 2.22(b)). After converting the regular expression to an NFA, pushing labels to the nodes, and incorporating the result into the query, one obtains the illustrated result (we use superscripts for new nodes that are not in the original query). The answer is essentially identical to this query.

As for the impact of this construction on the complexity, note that the conversion of a regular expression to an NFA can be done in polynomial time and space. Since the expression is part of the query, it means that the query may grow polynomially. The data complexity is unchanged.

**Example 2.6.4** Reconsider a variation on Example 2.5.7 (see Figure 2.23).

First, ignore the regular expression, then \(A_1\) is mapped to \(A\) in \(p_{1}^{S}\). The result derived from an embedding of \(p_{2}^{S}\) into a refinement of \(p_{2}^{S}\) is illustrated in 2.24(a). Now, assume the transitive edge from \(D_2\) to \(C_3\) is labeled with \((E.)^2E\), that is, three occurrences of \(E\) are requested, separated by occurrences of an arbitrary character. Then \(p_{2}^{S}\) is replaced as illustrated in 2.24(b). Here, the unnumbered new
nodes need to be mapped, but their images will be just the image of the original transitive edge.

Variables and joins Together with label predicates and regular path expressions, one may also want to use label and path variables and test for (in)equality of the assigned labels and paths. The interpretation is that query nodes labeled by (un)equal label variables are mapped to system nodes with (distinct)identical labels; query edges labeled by (un)equal path variables are mapped to paths whose sequences of labels are (different)equal words. While the use of label variables poses no particular problem, for queries with joins on path variables, our construction may fail; the answer to such queries may no longer be representable as a finite system.

To understand why, recall that our systems may be viewed as CFGGs. A query that tests for equality of path variables may have for an answer sets of graphs that are not a CFGG language and are inherently harder to compute, as illustrated by the following theorem. The theorem also highlights the difference in computational complexity between the querying of flat and nested graphs.

Theorem 2.6.5 For queries with equality conditions on path variables, the problem of testing whether the query answer is empty on a system is undecidable. The problem can be solved in exponential time if the system to which the query is applied has no recursive activities. It is PSPACE-hard even if the system BPs also have no cycles.
Finally, for flat BPs, the problem can be solved in time polynomial in the size of the system.

Proof: The undecidability and hardness proofs are by reduction to the problem of testing whether the intersection of the languages of two string context free grammars (CFGs) is empty. Given two CFGs $G_1, G_2$, we build a system $S$, with shape as in Figure 2.25. The implementation of the compound activity $R$ in the figure has two branches. The first contains a compound activity node $R_1$ and the second a compound activity $R_2$. The implementation of $R_i$, $i = 1, 2$, (which resembles in spirit the grammar rules of $G_i$ and is detailed below), is defined such that each of the paths from its start to its end nodes has line-shaped structure, representing a word in the context free language of $G_i$. The implementations are defined such that $R_1$ and $R_2$ can be refined to an activity sequence with the same shape iff this sequence represents a word that belongs to both $G_1$ and $G_2$. Next, we define the $q$ showed in Figure 2.25, with two transitive edges $e_1, e_2$ that match (the refinements of) $R_1$ and $R_2$ resp., and have an equality condition on their attached path variables. The query thus has a non empty result iff the languages intersection is not empty. This is known to be undecidable in the general case, and was recently proved to be PSPACE-complete for non-recursive context free languages [108].

To complete the undecidability and hardness proofs we need to explain how the implementation of $R_i$, $i = 1, 2$ is defined. We first consider the case where $G_i$’s are
general context free grammar, and explain later the case of non recursive ones. For a
case free grammar $G_i$, we model each of its non terminals as compound activity.
W.l.o.g. assume that $R_i$ is the root non terminal of $G_i$. The implementation of each
compound activity is defined according to the derivation rules of the corresponding
non terminal. It contains a start and an end nodes, both labeled by some label (letter)
that does not appear in the grammar. The start and end nodes are connected by
line-shaped paths, each representing one possible derivation rule of the non terminal.
(If the non terminal derives the empty word, the word the end and start nodes are
connected by a simple edge). Additionally we add, between every pair of connected
nodes, a “loop” with a node labeled by the new symbol $l$. To see a simple example,
Figure 2.26 shows the implementation of the compound activity for the a terminal $A$
deﬁned by the following derivation rules: $A \rightarrow cDA$, $A \rightarrow cA$, $A \rightarrow \epsilon$ (where $A, D$
are non terminals, $c$ a terminal and $\epsilon$ the empty string). The shaded part corresponds
to the derivation rules of $A$, while the remainder are the added loops.

It is easy to see that a word $w = a_1, \ldots, a_n$ belongs to the grammar $G_i$ iff some
refinement of $R_i$ contains a path of the shape $p = l^*a_1 l^* \ldots l^*a_n l^*$, starting (resp.
ending) at the start (end) node of $R_i$’s implementation. The number of $l$’s between
the word letters depends on the number of derivations steps performed to obtain $w$ -
each derivation step contributes at most two $l'$s between any two consecutive letters (due to the start and end nodes). An unbounded number of additional $l'$s can added via loop traversals.

Clearly, when some path $p$ appears in the refinements of both $R_1$ and $R_2$ (hence the result of the query $q$ described above is not empty) then the corresponding word $w$ belongs to both $G_1$ and $G_2$. Namely the languages intersection is not empty. To see that the converse also holds, note that if the derivation tree of word $w = a_1 \ldots a_n$ in $G_i$ is of depth $k_i$, then the refinement of $R_i$ contains, among others, all paths of the form $t^{k_i}a_1 t^{k'} \ldots t^{k_i}a_n t^{k'}$, for $k' \geq 2k_i$. This is because, as explained above, each derivation step contributes at most two $l'$s between any two consecutive letters, and the additional $l'$s need to reach a sequence of length $k'$, can be added via loop traversal. In particular, if $w$ belongs to both $G_1$ and $G_2$ than the refinements of both $R_2$ and $R_2$ contain a path $p$ where $k' = 2 \times \max(k_1, k_2)$, hence the query result is not be empty.

When the grammars $G_i$ are not recursive, the depth of the derivation trees for words in $G_i$ is bounded by a constant $d$. We use here a similar construction as above except that instead of plugging a loop (that allows to generate an unbounded number of $l'$s) between the graph nodes, it suffices to connect them with paths of varying length, containing between 1 to $2d$ nodes labeled $l$. This is illustrate in Figure 2.27 for $d = 1$. Note that the obtained BPs contain no recursion and no loops, and their size is polynomial in the size of $G_i$.

To conclude the proof we sketch the polynomial and exponential algorithms: For flat BPs, the algorithm considers all possible mappings of query nodes to the BP. Testing join conditions here amounts to testing if the intersection of the regular languages defined by the sub-graph that connects the nodes is empty, which can be done in PTIME. For nested, non-recursive, BPs, the algorithm enumerates all the system refinements (possibly an exponential number) and tests for the existence of a legal embedding in a similar way.

We have consequently decided to restrict the use of path variables in $\text{BP-QL}$ and allow joins only on label variables.
Distributed systems and queries  So far, we have ignored distribution. In a distributed setting, each peer holds a set BPs and may provides (resp. use) activities to (of) remote peers. If the service providers make their specification available to their cooperating organizations (say via a web service), users may wish to zoom-in on these remote components as well to query the service specification.

The data model extends naturally to this setting, associating a peer id with each process and each activity node. Queries may then annotate graph patterns and activity nodes by peer ids, restricting the search to the specified peers. In particular, when a (transitive) activity node in a query is annotated by a peer id, the search is restricted to implementations supplied by the specified peer (resp. refinements consisting only of invocations of activities of the specified peer). More generally, queries may use predicates on peer ids to restrict the search to a specific family of peers.

Remark: While the extension of the formal model to a distributed setting is rather immediate, implementation-wise, distribution poses significant challenges in terms of query evaluation. Specifically, we would like to evaluate a query in a “lazy” manner, so that only those peers whose processes and activities are indeed relevant to the query are consulted. Furthermore, it is desirable to “push” parts of the query, when
possible, to the peers holding the relevant process information. Our implementation, described in the next section, addresses these issues.

**Summary** The design of BP-QL was directed by the special requirement of querying specifications with a zoom-in feature at different levels of granularity and the retrieval of qualifying execution paths. As explained above, this required a careful design of the language to avoid features that might seem to be worthy of inclusion in the language, such as joins on path variables, but incur a prohibitively high computational cost.

The characterization of the exact expressive power of BP-QL is an on-going research. Our initial results indicate that BP-QL can be characterized as a particular subclass of FO(TC)\(^7\). In particular, for flat BPs BP-QL captures power similar to that of the conjunctive part of XPath and core XQuery, including negation, when considered in the context of graphs.

### 2.7 Implementation

The query language presented above has been fully implemented and tested in the BP-QL peer-to-peer system. The system provides persistent storage for BPEL specifications, allows users to design new processes, and to query existing specifications.

The visual interface of the system is implemented as an Eclipse [61] plug-in. It allows to: design new business processes and store their specifications in the repository; import existing BPEL documents to the repository; formulate queries, run them and view the results. The rest of the section is devoted to the main component — the query engine.

#### 2.7.1 Design Considerations

BP-QL is based on an intuitive, conceptual model of BPs, an abstraction of the BPEL specification, allowing for simple formulation of queries over this model. When we considered the implementation, the following problem had to be addressed: As mentioned in Section 2.1, the BPEL XML format was designed with ease of automatic

\(^7\)First Order Logic augmented with Transitive Closure.
code generation, rather than querying, in mind. Activities and edges are defined separately, as distinct activity and link elements. The process flow is only recorded by associating with each activity element the ids of its incoming and outgoing edges, represented resp. by target and source children of the node. This is illustrated in Figure 2.28, which shows the BPEL XML representation of the Travel Agency business process from Figure 2.1. Consequently, to check whether flow paths of a given process satisfy the conditions detailed in a BP-QL query, a large, possibly unbounded, number of join operations involving edge ids between activity and edge elements needs to be performed. While this is expressible in, say, XQuery, e.g. with the use of recursive functions, the excessive number of joins becomes a performance bottleneck.

To drastically reduce the number of joins, we decided to store a process specification in a structure more similar to its graph view. In XML terms, the parent child relationships in the XML representation of a process should reflect the “followed by” relationship of nodes in the process graph. This would allow the use of XPath’s ”/” and ”//” operators for querying flow paths, avoiding many joins. But, since a typical BP is a graph, rather than a tree, we also use XML idrefs to capture the graph structure.

Another fundamental decision to be made was which of the following two options

---

8For simplicity, the figure provides an abstraction of the actual BPEL XML file structure, with many details omitted.
2.7. IMPLEMENTATION

to choose: (1) to implement a whole new query engine for our model from scratch, or (2) to rely on some existing query engine to perform as much as possible from the computation, and complete the processing of the missing features by an adequate pre and post processing of queries and query results. We opted for the second option. The issues to be considered in selecting an engine were the following:

- Our query language allows to retrieve paths, whereas typical existing XML/graph query languages only retrieve nodes.
- Our query language offers a zoom-in facility.
- Business processes typically operate in a cross-organization, distributed environment. The specifications of the services participating in process may reside on distinct peers. Distributed query processing thus becomes essential.

A natural candidate was to use a standard XQuery engine, enjoying the benefits of indexing and optimization offered by such engines. However, XQuery does not support the retrieval of paths, distribution, or zoom-in queries; nor does it “traverse” idrefs. Necessarily, all of these would have to be implemented by pre and post processing. Consequently, we decided to base our solution on an extension of XML, called Active XML (AXML for short). AXML is essentially a middleware system that includes an XQuery-like query language, but offers additional facilities which provide better support for addressing some of the above issues. Additional benefits include certain optimization techniques that are implemented in the AXML system, as explained below.

A brief overview of Active XML Active XML (AXML, for short) is a declarative framework that harnesses Web services for data integration, and works in a peer-to-peer architecture[5]. An AXML document is an XML document where some data is given extensionally, as regular XML elements, while other data is given intensionally, by means of calls to Web services[5], and can be materialized by invoking the services. AXML employs the query language XOQL, an XQuery-like query language as its query engine. When a query is evaluated on an AXML document, the service calls

\[9\] An alternative viable solution to the graph shape of BPs could be to use a native graph query engine.
whose answer *may be relevant* for the query are identified; only these calls are invoked. Additionally, (sub-)queries are *pushed*, when possible, to the service providers, thus reducing the costs of data materialization and transfer. Recursive calls are tracked, and only the relevant data is materialized (see [2] for details).

In summary, **BP-QL** uses the AXML system [5] as an implementation platform. The facilities offered by AXML are used to address our needs, as follows: Intentional data, implemented by service calls, are used in our implementation to (1) retrieve, when needed, the specifications of remote processes, thus supporting distributed processing, and (2) account for the graph structure of the specification (service calls play here role similar to XML idrefs, with the advantage that they are traversed automatically in query evaluation). BPEL documents are wrapped and represented as AXML documents; **BP-QL** queries are pre-processed and compiled into a set of XQuery-like queries over such documents. Post processing is employed to complete the computation, e.g. to validate zoom-in relationships, to extract paths and to construct a compact representation for the result.

**From BP-QL to AXML**  Here is a brief description of the AXML representation of a **BP-QL** business process. The representation consists of three parts: Process properties (such as the service provider, the service type and capabilities) are maintained as UDDI entries in a (standard) XML document. The other two, namely the process activities and execution flow, and the data elements and the data flow, are maintained in two AXML documents. The use of two AXML trees, rather than one, allows for efficient evaluation of **BP-QL** queries with double headed edges: it allows a doubly headed activity (resp. data) flow edge to be mapped to a “//” operator on the corresponding AXML document.

For example, Figure 2.29 describes (part of) the AXML tree for the Alpha-Tours activities and flow. (Here again, for simplicity, only an abstraction of the actual AXML tree is provided, with many details omitted.) Each activity is represented by an XML element node in the tree. The parent child relationships reflect the flow. Each node representing a compound activity is the root (labeled by *zoom-in*) of a subtree that describes the internal structure of the activity. Nodes with bold labels
2.7. IMPLEMENTATION

Figure 2.29: AXML tree.
are special elements that represent calls to Web services. Two types of such calls are embedded in the document:

- A `getActivity` service call plays a role similar to that of an XML `idref`, “pointing” to a certain node in the tree. When a query is evaluated, the relevant calls are detected and invoked. (Cycles are detected and cut by AXML). For each call, the returned data (a copy of the sub-tree “pointed to”) is inserted in place of the service call, ready to be accessed. Thus query evaluation can access the returned subtree as if it actually traversed the “pointer”.

- A `getOperation` service call retrieves the specification of a remote compound activity and converts it, when needed, from BPEL format to an AXML representation. A zoom-out element is attached to its final state, so that it points to the following activity in the flow.

To illustrate the first type of call, the `getActivity("join")` nodes below the `searchFlights` and `searchRooms`, in the middle of Figure 2.29, point to the `join` node below `searchCars`. They represent the fact that the three searches are followed by that same `join` operation.

The `getOperation("searchRooms","join")` in the figure illustrates the second type of call. It retrieves the specification of the `searchRooms` process, and set its zoom-out to the following “join” operation. Here again, AXML invokes `getOperation` calls for the remote activities whose specification is judged to be relevant for query evaluation. As mentioned above, it may also “push” (sub-)queries to capable service providers, such as BP-QL peers that “understand” BP-QL queries.

Data elements and data flow are represented in AXML tree in a similar manner: The tree contains both data and activity element nodes. `getData` and `getActivity` service calls are used as “references” between tree data and activity nodes, resp.

To generate the AXML representation, the BP–QL graph is traversed in a depth-first order, building AXML trees as deep as possible. Local compound activities

---

10 In the implementation, the input to the `getActivity` is a unique identifier for the pointed activity, consisting of BP and activity ids. It is abstracted here, for brevity, by the activity name.
are then zoomed-in and their graphs are similarly detailed, recursively. Requests to remote operations are represented by \texttt{getOperation} service calls. Web services are generated for provided operations, exposing their specification to the requesting peers.

With this representation, both the path-based and the zoom-in axis conditions can be evaluated using XOQL queries on the AXML documents. Some post processing is nevertheless required to match up the components, extract the requested paths (XOQL, like most XQuery engines, returns only document elements not paths), and construct a compact representation of the result.

\subsection{2.7.2 Trade-offs}

As explained above, we have decided to store BPs in a structure close to the BP graph shape, rather than in the BPEL format. Obviously, this reduces the number of join operations required in query evaluation. With this representation, it is still necessary to account for the graph structure of BPs. This can be taken care of by performing joins. Instead, the use of AXML allows to represent “cross edges” by service calls. The price payed for this, performance-wise, is the invocation of service calls: For example, \texttt{getActivity} calls are invoked when “pointers” need to be traversed.

To understand the trade-offs, we performed the following experiment. We considered BPs with varying depth and width, where depth is the maximal length of (simple) paths from the start node to the end node of a BP; and width is the maximal in-degree of nodes in its graph. They reflect, resp., the number of joins saved by moving from a “flat” BPEL format to the hierarchical representation, and the number of service calls that may be invoked when “traversing pointers” to a given node. We selected as a representative class of path-oriented queries those that search for the occurrence of a given activity, followed (at an arbitrary distance) by another given activity. All the tests were performed on IBM Laptop T43, 1.86 Ghz, 1Gb of RAM with Windows XP, sp2. A representative sample of results is shown in Figures 2.30. The BP graphs here include a fork activity that splits the flow into 5,7,10,12 and 15 different paths that are joined later, and followed by a tail of length 1,3,5 and 7 (on the x-scale).
We measured the respective evaluation time of the (translated) BP-QL queries on the AXML and BPEL representations of the BPs. The AXML result columns are presented in front of the BPEL columns. For clarity, the figure shows only the net query running time; the time of Web service calls is excluded from AXML columns. By our measures, an average `getActivity` service call takes about 100msec. AXML performs most calls in parallel, so the typical overall delay due to the materialization of data is also around this number.

As we can seen, the running time of queries (for both BPEL and AXML) grows linearly with the BP width. (For BPEL, this is because more nodes participate in the joins. For AXML, this is because the "//" has more paths to traverse.) For narrow graphs, although the use of our representation reduces the number of joins, the relative overhead of service calls is substantial. The relative benefit of using our representation and AXML over using the BPEL representation for wider graphs grows with the BP depth. For depth greater than 7 (values larger than 7 are omitted from the figure), the gain from the saving of joins outweighs the additional cost of data materialization via service calls.

While the use of Web services brings some (moderate) overhead to query processing, it allows for greater flexibility in distributed data processing. To see if (and how) the distribution of data affects query processing we performed the following experiment. We considered business processes consisting of several compound activities, and varied the number of peers that hold the specifications of activities. At one extreme, the full specification resides on a single peer. At the other extreme, each
process activity is provided by a distinct peer. We compared the execution time of queries on these varying configurations, considering both global queries (that consult the specifications on all peers) and local queries (where the search is restricted to only local specifications.) Figure 2.31 illustrates a representative sample of the results. It considers the Travel Agency from our running example, and the query from Figure 2.4 (with the search scope set to local and global, resp.). We varied the number of local compound activities (operations whose specifications reside on the local machine) from one to all (5), moving the remaining specifications to remote peers. We see that the cost of the global queries is practically independent of the distribution level. Not surprisingly, the execution time of the local query increases linearly as more portions of the BP are local, since more data is available for querying.

2.8 Summary

We presented BP-QL, a novel graphical Query Language for querying Business Processes. BP-QL allows users to query business processes visually, in a manner very close to how such processes are typically specified, and can be employed in a distributed P2P setting.

The language provides flexible granularity: it allows to navigate between two axis, along the flow and zooming into activities. It also supports path retrieval and operates in a distributed environment. The theoretical model is graph grammars and
this enable us to represent compact answers that look like the system.

We ignore semantics of some BPEL constructs and data values. This is a reasonable tradeoff between expressivity and complexity. The use of AXML as an implementation platform supports transparent distribution and takes advantage of built-in optimization.

The BP-QL language is based on an intuitive model of business processes, an abstraction of the emerging BPEL (Business Process Execution Language) standard [23]. Other previously proposed standards like [60, 25, 48] can similarly be supported, exploiting the abstraction level of our formal model.
Chapter 3

Monitoring Business Processes with BP-Mon

3.1 Introduction

A BPEL spec is compiled to code that runs on a BPEL application server. This server is a software platform that facilitate the definition, deployment, execution, and monitoring of BPs. Because of their central role in carrying out business activities, and their complexity, monitoring of BPs is a critical activity in modern enterprises. In this chapter we focus on this important aspect of such systems, namely the monitoring of processes execution.

For some intuition about the type of monitoring that a BP may require, consider a simple example. Imagine a manager of a Web-accessible auctioning business. Monitoring of process executions may allow the manager to guarantee fair play, detect frauds, and track services usage and performance. The manager can ask, for instance, to be notified whenever an auctioneer cancels bids too often, or when buyers attempt to confirm bids without first giving their credit details, so that she can block their actions. Similarly, being notified whenever the average response time of the database in a given service passes a certain threshold, allows her to fix the problem or switch to a backup database. In general, monitoring encompasses the tracking of particular patterns in the executions of individual processes or in the interaction between
different processes, as well as the provision of statistics on the performance of some processes or the system. *Our goal here is to provide intuitive, easy to use tools to facilitate this critical task.*

Before presenting our query language, we briefly highlight some of the main characteristics of existing BP Management Systems and the challenges encountered in monitoring current BPs.

**Background** As explained previously, BPEL specifications are typically defined via a visual interface, then compiled into code that runs on a BPEL application server. An *instance* of a BP specification is an actual running process which includes specific decisions, real actions, and actual data. BP Management Systems allow to trace process instances – the activities they perform, messages sent or received by each activity, variable values, performance metrics – and send this information as events (in XML format) to *monitoring* systems (often called BAM – Business Activity Monitoring – systems). Typical monitoring systems (e.g. [14, 81]) allow users to specify events of interest, and actions to be performed when the events are identified. Events may be atomic or composite (i.e. consist of a group of other atomic or composite event). Detection and processing of (composite) events has been an active research area since the early 90’s. Rich event algebras have been proposed for describing composite events (e.g. [72, 116]), and sophisticated evaluation and optimization techniques have been developed for their detection [95] (see Section 5.2 for details). Nevertheless, existing technology suffers from three main drawbacks when it comes to the monitoring of BPEL BPs.

**Abstraction level** In existing systems, the specification of monitoring tasks and, in particular, of the the relevant (composite) events, requires intimate knowledge of both the monitored application and the specific events emitted by activities. This is contradictory to the high level abstraction employed when *defining* BPEL BPs, where implementation details (including the types of run-time events generated by the system) are deliberately hidden. Thus, programmers nowadays use two distinct tools, one for defining BPs at a high level of abstraction, typically via a graphical UI, and another for defining monitoring tasks for the BPs, typically via lower level Event-Condition-Action style rules. The abstraction gap between these tools is akin
to the one between assembly and high level languages.

To close this gap, it is desirable that the specifications of monitoring be performed on the same (high) level of abstraction as that of the BPs, possibly even using a similar specification language. Such a monitoring tool would allow (a) simultaneous formulation, by the BP designer, of a BP and its corresponding monitoring tasks, and (b) a faster learning curve of the monitoring language.

**Optimization**  A variety of methods have been proposed for optimized processing of (composite) events, employing relation and object-oriented database technology [132, 72], petri nets, finite state automata, event graphs, and storage minimization (See [116] for a survey). The proposed methods are generic, hence can be employed in a variety of application domains, including BP monitoring. A disadvantage, however, of a generic approach is that it does not exploit the particular properties of BPs and available knowledge about them. As a simple example, assume we wish to be notified when a given activity sequence occurs in some process. If, according to its (BPEL) specification, an activity $o$ never co-occurs with such a sequence, monitoring for the sequence can be stopped immediately if an activation of $o$ is detected. While such knowledge about BPs structure is naturally valuable for optimization, to our knowledge it is not exploited by any current BP monitoring tool.

Runtime monitoring that considers the processes structure has been studied, e.g., for models and query languages based on temporal logics as LTL (See Section 5.2). But what is desirable here are optimizations stated directly in terms of BPEL and a corresponding high-level monitoring language.

**Deployment**  As mentioned above, BPs are specified in a high level manner and the specifications are automatically compiled into executable code that can, in principle, run on any BPEL application server [112]. Analogously, it is desirable that a monitoring task be defined in a declarative manner, and be compiled, and easily deployed, on whatever BPEL application server chosen for the monitored BP. In existing monitoring tools, however, the monitoring tasks are written in proprietary languages and are not portable[67].
The **BP-Mon** (BP Monitoring) system presented here addresses these three problems, making the following contributions.

**Query language** We present a high-level intuitive graphical query language that allows for simple description of the execution patterns to be monitored. A tight analogy between the graphical interface used by commercial vendors for the specification of BPEL BPs and the graphical query interface that we use for monitoring allows natural and intuitive design of monitoring tasks.

**Evaluation and optimization** We provide a dedicated efficient automata-based algorithm to identify occurrences of monitored patterns. We present a novel optimization technique that speeds up computation, by pruning redundant monitoring, based on an analysis of the process BPEL specification.

**Implementation and deployment** To support flexible deployment, our system compiles a **BP-Mon** query $q$ into a BPEL process specification $S$, whose instances perform the monitoring task. As for all standard BPEL specifications, $S$ can now be automatically compiled into an executable code to be run on the same BPEL application server as the monitored BP. We describe experiments that indicate that the resulting monitoring is very efficient and incurs only very minimal overhead.

In summary, **BP-Mon** offers a high-level, intuitive design of monitoring tasks. It compiles these tasks into efficient and standard BPEL processes, thus providing easy deployment, portability, and minimal overhead.

**Discussion** In Chapter 2 we proposed to use a graphical query language for querying BP specifications. There, the goal was to be able to retrieve specifications with certain properties (e.g. where an execution path from activity A to activity B is possible), and the solution relied on modeling specifications and queries as graph grammars. In contrast, our work here is concerned with querying the actual execution of process instances (e.g. to find when an actual execution path that started at activity A arrives to activity B), and the solution is based on automata construction. The two works are complementary: monitoring can be used to check at runtime properties that cannot be statically determined by querying the specification, while querying the specification can be used to discover parts of BPs that require monitoring.

As mentioned above, events are sent to monitoring systems in XML format. A
natural question is why not use XQuery, coupled with some XML stream-processing engine [86], to process this stream? A key observation is that the XML elements in this stream describe individual events. To express any non-trivial query about a process execution flow, one needs to write a fairly complex XQuery query, that performs an excessive number of joins, and can hardly (if at all) be handled by existing streaming engines. Furthermore, standard XML stream processing would still be inadequate for the task, even if a more query-friendly nested XML representation, that reflects the flow, had been chosen for the data. XML stream engines manage tree-shaped data, expect to receive the tree elements in document order, and process siblings sequentially, as they arrive. However, a BP execution is essentially a nested set of DAGs. In a DAG, some activities may run in parallel and interleave, hence the events flow in BPs does not necessarily follow document order. Nesting of DAGs in BPs follows from the fact that processes contain composite activities with complex internal execution flow, itself represented by a DAG. Interleaving of events from different DAGs of a BP is another aspect of parallelism. Here, parallel processing, that processes each event according to its position/nesting in the flow is called for. This is provided by BP-Mon.

Chapter organization  The remaining of this chapter is structured as follows: Section 3.2 provides an overview of BP-Mon, illustrated by examples. Section 3.3 describes the basic formal model underlying BP-Mon’s execution patterns. Section 3.4 deals with query evaluation and optimization. Extensions to the model are considered in Section 3.5. Section 3.6 describes our implementation and the experiments performed to measure the performance of the system, and we summarize this chapter in 3.7.

3.2 Monitoring Business Processes

We present here an informal overview of BP-Mon via a running example that extends the Web auctioning BP scenario introduced in Section 3.1.
3.2.1 Underlying Technology

Let us first briefly describe some of the underlying technology; what BPEL BP instances are and what data is available for their monitoring. Throughout this chapter we describe the data model and query language using an auction service example.

**BPEL** The BP in Figure 3.1 illustrates the main flow of the *auctionHouse* BP. The circle at the top of a BP (see Figure 3.1) is its entry point; the square at the bottom is its exit point. In the BP of Figure 3.1, users register to the system by invoking the *register* activity (whose details are not shown here). As part of this activity, they
choose to play a seller or a buyer role. Depending on their choice, they are directed in
the following switch activity to one of the two compound activities: seller_process and
buyer_process. Figure 3.2 is a zoom-in into the seller process, that shows its internal
flow.

While in Chapter 2 which describes an abstraction of the BPEL specifications
we used our own notation, here we use the BPEL for the different activity types. Activities
that are invoked by (resp. invoke) other activities/users are marked by small incoming
(outgoing) arrows. The BPEL switch, while, and flow constructs are represented by
diamond shaped nodes that contain a question mark, a circular arrow, and two parallel
lines, respectively. The switch icon in Figure 3.1 was explained above. The while icon
at the top of Figure 3.2 indicates that the seller can repeat the described
activity any number of times. At each round she can either manage her existing
auctions (e.g. decide to cancel the whole auction, decide to cancel some specific
user’s bid, etc.) or add new items to sale. New items are added to the database by
the add_item_to_db activity. Once the update is confirmed the track_action process
is invoked to wait until the auction ends and declare the winner. The internal structure
of this process is depicted in Figure 3.3. The flow construct here indicates that the
winner and the seller are handled in parallel. The process notifies them about the
auction results and awaits their approval.
(actionData)
    (header)
        (processName) auctionHouse (/processName)
        (instanceId) 517 (/instanceId)
        (sensor target="add_item_request" /)
        (timestamp) 2006-05-31T11:32:46.510+00:00 (/timestamp)
    (/header)
    (payload)
        (activityData)
            (activityType) receive (/activityType)
            (evalPoint) completion (/evalPoint)
            (durationInSeconds) 0.1 (/durationInSeconds)
        (/activityData)
        (variableData)
            (target) $itemVar (/target)
            (data) (addItemRequest)
                (category) MP3 player (/category)
                (description) iPod mini 4GB (/description)
                (price) 50 (/price)
            (/addItemRequest)
        (/data) (/variableData) (/payload)
(/actionData)

Figure 3.4: BPEL event.

<table>
<thead>
<tr>
<th>sensor@target</th>
<th>evalPoint</th>
<th>timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait_auction Ended</td>
<td>activation</td>
<td>2006-05-30 T 08:00:00.510</td>
</tr>
<tr>
<td>wait_auction Ended</td>
<td>completion</td>
<td>2006-05-31 T 11:32:42.000</td>
</tr>
<tr>
<td>flow</td>
<td>activation</td>
<td>2006-05-31 T 11:33:00.550</td>
</tr>
<tr>
<td>notify_winner</td>
<td>activation</td>
<td>2006-05-31 T 11:34:44.220</td>
</tr>
<tr>
<td>notify_seller</td>
<td>activation</td>
<td>2006-05-31 T 11:35:05.670</td>
</tr>
<tr>
<td>get_seller_conf</td>
<td>activation</td>
<td>2006-05-31 T11:35:46.450</td>
</tr>
<tr>
<td>get_seller_conf</td>
<td>completion</td>
<td>2006-05-31 T 11:35:00.550</td>
</tr>
<tr>
<td>notify_winner</td>
<td>completion</td>
<td>2006-05-31 T 11:36:02.210</td>
</tr>
<tr>
<td>get_winner_conf</td>
<td>activation</td>
<td>2006-05-31 T 11:41:26.530</td>
</tr>
<tr>
<td>get_winner_conf</td>
<td>completion</td>
<td>2006-05-31 T 11:39:27.010</td>
</tr>
<tr>
<td>flow</td>
<td>completion</td>
<td>2006-05-31 T 11:39:48.010</td>
</tr>
</tbody>
</table>

Figure 3.5: Add item audit stream.
3.2. **MONITORING BUSINESS PROCESSES**

**BPEL events** An *instance* of a BPEL specification is an actual running process that follows the logic described in the specification. BP Management systems allow to trace process instances. For each activity issued, two events are generated, at its activation and completion, respectively. Events are reported in XML format. Figure 3.4 shows a completion event for the *add_item_request* activity of Figure 3.2 (with some data omitted for brevity). The header includes identification information for the event: the BP name, the instance ID, the activity name, and a time-stamp. The provided data includes the activity type (e.g. *invoke*, *receive*, *sequence* etc.), the reporting point (activation or completion of the activity), the activity duration, and variables information (variable names and values).

For a compound activity, the events corresponding to its internal flow are reported between its activation and completion events. The events stream of an instance can be viewed as a (nested) DAG (see Figure 3.6). The nodes for an activity represent its activation and completion events, resp. *Flow* edges (represented in the figure by solid arrows) connect activation and completion nodes of the same activity and record causal dependencies between distinct activities of a process. *Zoom-in* edges (represented by dashed arrows) connect the activation (resp. completion) node of each compound activity to the the start (rep. end) nodes of the DAG that describes the activity’s internal flow. Note that the edges in the DAG connect nodes with increasing time stamps. Recall that some activities may run in parallel (e.g. *notify_winner* and *notify_seller*). At any given time *t*, the DAG represents the execution up to point *t*.

The stream of events of a given instance can be viewed as a (nested) DAG (see Figure 3.6). Each activity is represented by a pair of nodes, for its activation and completion events, resp. Two type of edges are used: *Flow* edges (represented in the figure by solid arrows) connect activation and completion nodes of the same activity and record causal dependencies between distinct activities of a process. *Zoom-in* edges (represented by dashed arrows) connect the activation (resp. completion) node of each compound activity to the start (rep. end) nodes of the DAG that describes the activity’s internal flow. \(^1\) \(^2\) It should be noted that the edges in the DAG connect

---

\(^1\)The causality and zoom-in relationship between events may be given as part of the events data or can be inferred from the BP specification. This is discussed in more details in Section 3.6.

\(^2\)The causality and zoom-in relationship between events may be given as part of the events data
nodes with increasing time stamps. At any given time $t$, the DAG represents the execution up to point $t$.

### 3.2.2 BP-Mon

In the auction scenario, the system supervisor may want to be notified when a seller cancels bids or auctions too often, or when buyers attempt to confirm bids without giving their credit details first. She may also want to be informed when the average response time of the database server for a given service passes a certain threshold and to gather statistics about the response time. **BP-Mon** queries can be used to monitor process instances and accomplish these tasks.

For monitoring process instances, **BP-Mon** uses execution patterns (abbr. EX-patterns). Intuitively, these extend string regular expressions to (nested) DAGs. The patterns look much like the specifications. In addition to standard BPEL constructs, such as *while*, *switch*, etc., they may include two additional new constructs, denoted

(e.g. the BPEL “correlation set” for *invoke* and *receive* activities) or can be inferred from the BP spec.
or and rep, describing, resp., alternative patterns and repetitions. The patterns also allow to navigate in the activities flow along two axes: path-based and zoom-in-based. Following the use of / and // in XPath[130] to denote single and multiple step navigation, our patterns use edges with single and double heads to denote single and multiple edge paths, resp. Similarly, to allow users to query about activity flows that are nested at any depth in the zoom-in hierarchy, compound activities may have doubly bounded boxes, denoting an unbounded zoom-in into the activities internal flow.

The activities and edges of EX-patterns can be associated with variables, which can be used in selection conditions on the values of the associated attributes/data variables and in reports. To issue a report, a reporting icon, depicted as a page with two small arrows, can be connected to a reporting point in the pattern (an atomic or a compound activity). A BP-Mon query may include several such reporting icons/points. Two reporting modes are available: local, where an individual report is issued for each process instance, and global, that considers all the BP instances. For each report one can specify when should it be issued (e.g. at the first time that the reporting point is reached, at periodic time interval, or when certain conditions are satisfied) and what should be the structure of the output (in XML format) or the actions triggered at this point. The following examples illustrate the features available for monitoring.

Example 3.2.1 The query in Fig. 3.7, monitors auctions to guarantee fair play. It looks for users that register as sellers, and repeatedly cancel bids or auctions. The ‘or’ here denotes that the pattern that we look for is an occurrence of one of the two cancel activities. The ‘rep’ denotes repetitions of this pattern, with the \( \geq 5 \) indicating that at least 5 occurrences are required. The double headed arrows indicate that the activities may have occurred at any distance from the beginning of the seller process, and also at any distance from each other (in the given instance). The double bounding of the seller process box denotes unbounded zoom-in; we look for cancelation activities in this process and (transitively) the compound activities that it includes/invokes. A report, with the name of the corrupt auctioneer, is issued as soon as the pattern is matched, i.e. when five cancelations are identified. If we want to get re-notified
Figure 3.7: Too many cancels.

Figure 3.8: Average response time.
if/when cancelations are further repeated, a Report* command can be used instead. Finally, to trigger corrective activity, an Invoke command is used.

**Example 3.2.2** The query in Figure 3.8 may be used to guarantee service quality. Assume that the datastore service is in charge of interaction with the database and is used massively in the auctioning process to store and manage items and bids. We would like to monitor its response time. We look for a pattern of an invoke activity, immediately followed by a receive operation. (The partnerlink attribute identifies the target/source service). Note that the single headed arrow here indicates consecutive operations. Also note that we use here a global reporting mode that aggregates the data of all the BP instances. Let us now consider the some types of reports. The following is an example for a time-based sliding window report, where we request to get an hourly report of the average response time and standard deviation in the last couple of hours:

```
Report* Every 1 hrs Range 2 hrs
<response-time>
  <avg>
    {avg($y/activation/timestamp-$x/completion/timestamp)}
  </avg>
  <std>
    {std($y/activation/timestamp-$x/completion/timestamp)}
  </std>
</response-time>
```

BP-Mon also supports match-based windows where the window slides over the previous matches of the pattern in the given instance (if the report is local) or in all the running instances of the given BP (if it is global). For example, to issue a report, every 100 matches, that provides the average response time and standard deviation of the last 200 calls to the database, we could use in the above report

```
Every 100 entries Range 200 entries
```

Grouping may also be employed. For instance, we can group the database calls according to the type of the requested operation and report response time only for operations with frequency $\geq 10$. For that we add at the bottom of the above command
Example 3.2.3 Assume that, to promote sales, we wish to periodically give prizes to our users. For instance, we may want to credit the seller and buyer pairs that participated in the 10000 sell, 20000 sell, 30000 sell, etc. The monitoring query in Figure 3.9 reports the names of the winners. Since notifications for each buyer-auctioneer pair are processed in parallel (recall the flow construct in figure 3.3), so is their monitoring.

Example 3.2.4 Finally, the query in Figure 3.10 monitors illegal access. It identifies instances where a user attempts to submit bids without first registering to the system, and reports the instance ID and the corrupt execution path. We use here a path predicate (essentially a subquery) that is attached to the transitive edge connecting the start node to the bid_request node and restricts the assigned paths to those that do not include registration.

We may furthermore want to combine run-time monitoring with specification analysis, and identify execution paths that do not comply with the BP specification. The query in Figure 3.11 compares a bidder’s run-time execution flow (the $x$ on the left) to what is allowed according to the specification (the $y$ on the right). In this simple example the two query patterns, on the run and the specification, are similar, but in general one can use different patterns, e.g. one pattern on the specification to identify what needs monitoring, and another pattern on the run to perform this monitoring.

The queries so far all have a single reporting point. We also support queries with multiple reports.

Example 3.2.5 Assume that we wish to obtain weekly statistics about the average age of the users that register to the system at different times of the day. We can attach to the register node of the query in Figure 3.7 the following report request.

Report Every 1 week Range 1 week
<age-by-hour>
<avg>{avg($x/data/age)}</avg>
3.2. MONITORING BUSINESS PROCESSES

Figure 3.9: x10,000 buyer.

Figure 3.10: No registration.

Figure 3.11: Static and dynamic analysis.
GROUP BY $x/activation/timestamp/*:hour(.)

Here is a comment about the semantics of such report points (a full account is given in Sections 3.3 and 3.5). As mentioned earlier, a user may request a report to be issued as soon as a match for the pattern is identified. In order not to block reporting, only the nodes and edges in the pattern that precede a report point are considered relevant to it. For a report to be conditional on the occurrence of the full pattern it needs to be attached to the last node in the pattern or to the outermost box, as in all the previous examples. Thus, the report here will include information about all registered users, regardless of whether they had later canceled bids or not. To get the same reports only for corrupt users, the same report should have been connected to the rep or the auctionHouse boxes.

3.3 The Formal Model

BPMQL queries consist of two main ingredients: (1) EX-patterns that are matched to execution traces and (2) reports generated from these matches. Reports are discussed in Section 3.5; we focus here on BP-Mon’s pattern matching. We first explain how execution traces are modeled and then consider EX-patterns and their semantics. (An efficient algorithm to identify pattern occurrences is presented in section 3.6). To simplify the presentation we consider in this and the next section a basic data model. We then enrich it in Section 3.5 to obtain the full fledged model.

Event traces As mentioned earlier, the execution trace of a process instance can be viewed as a DAG. Each activity is represented by a pair of nodes, corresponding to its activation and completion. When an activity is compound, the DAG that represents its internal flow appears (time-wise) between the activity activation and completion nodes, and is connected to them by zoom-in edges. This is formalized below.

We assume the existence of three domains, $\mathcal{N}$ of nodes, $\mathcal{L}$ of node labels, and an ordered domain $\mathcal{T}$ of time stamps. We first define the auxiliary notion of activation-completion labeled DAGs.
3.3. THE FORMAL MODEL

Definition 3.3.1 An activation-completion labeled DAG is a tuple $G = (N, E, \lambda, \tau)$ in which $N \subset \mathcal{N}$ is a finite set of nodes, $E$ is a set of edges with endpoints in $N$, $\lambda : N \to \mathcal{L}$ is a labeling function on the nodes, and $\tau : N \to \mathcal{T}$ is a time-stamp function on the nodes. We assume $G$ satisfies the following:

1. The edges in $E$ are of two types: flow, and zoom-in.
2. The nodes in $N$ are partitioned into pairs, called activity pairs. Each pair $n_1, n_2$ shares a label, i.e., $\lambda(n_1) = \lambda(n_2)$. In such a pair, one node is designated as an activation, the other as a completion; they are denoted by $\text{act}(l)$ and $\text{com}(l)$, resp., where $l$ is their shared label. There is precisely one flow edge from $\text{act}(l)$ to $\text{com}(l)$; no other flow edges leave $\text{act}(l)$, and no other flow edges enter $\text{com}(l)$.
3. $\tau$ assigns distinct time stamps to nodes of $G$ s.t. if there is an edge from $n_1$ to $n_2$, then $\tau(n_1) < \tau(n_2)$.

We assume the graph has a single start node without incoming edges, and a single end node without outgoing edges, denoted by $\text{start}(G)$ and $\text{end}(G)$, resp.

Definition 3.3.2 The set $\mathcal{EX}$ of execution traces (abbr. EX-traces) is the smallest set of graphs that satisfies the following.

1. [flat trace] If $G$ is an activation-completion labeled DAG without zoom-in edges, then $G \in \mathcal{EX}$.
2. [nested trace] If $G_1, G_2$ are in $\mathcal{EX}$, and $(\text{act}(l), \text{com}(l))$ is an activity pair of $G_1$, then the graph $G$ consisting of $G_1$, $G_2$, and two new zoom-in edges $(\text{act}(l), \text{start}(G_2))$ and $(\text{end}(G_2), \text{com}(l))$, is in $\mathcal{EX}$, provided the combined time-stamp function $\tau$ on $G_1 \cup G_2$ satisfies constraint 3 of Definition 3.3.1 above.

A prefix of an EX-trace is defined in the standard way, as any graph obtained by removing some nodes, all their descendent nodes, and all edges into and out of deleted nodes.

In the sequel, we call a subgraph $G_2$ that is connected, as in Item 2 above, by zoom-in edges to an activity pair $(\text{act}(l), \text{com}(l))$, an internal trace of the pair. Such a subgraph, omitting the internal traces of its own activities, is called a direct internal
trace. Observe that in general a given activity pair may have several internal traces connected to it. This happens when the activity implementation includes several parallel processes.

Queries are modeled by execution patterns (abbr. EX-patterns), that generalize EX-traces similarly to the way tree patterns generalize XML trees. EX-patterns are EX-traces without time stamps (since they are not real executions but just patterns) where node labels are either specified, or left open using a special any symbol, and where two additional new label symbols can be used: or and rep. or describes alternative patterns and rep describes one or more repetitions of a given pattern. Edges in a graph are either regular edges, interpreted over edges, or transitive, interpreted over paths. Similarly, activity pairs may be regular or transitive, for searching only in their direct internal trace or zoom-in transitively inside it.

**Definition 3.3.3** An execution pattern (EX-pattern) is a pair $p = (\hat{e}, T)$ where $\hat{e}$ is an EX-trace without time stamps\(^3\), whose nodes are labeled by labels from $\mathcal{L} \cup \{\text{any}, \text{rep}, \text{or}\}$, and $T$ is a distinguished set of activity pairs and flow and zoom-in edges in $\hat{e}$, called transitive activities and edges, resp. The nodes in $p$ with labels other than rep and or are called concrete nodes. We say that $p$ is concrete if it contains only concrete nodes.

Graphical BP-Mon queries are naturally mapped to EX-patterns: Each activity icon labeled $l$ is mapped to a pair of nodes $\text{act}(l), \text{com}(l)$, that inherit properties like double bounding. Additionally, nested activities are connected by zoom-in edges (simple or double-headed) to these two nodes in the obvious manner. For example, Figure 3.12 depicts the EX-pattern corresponding to the query of Figure 3.7. (The reporting part is omitted for now and will be considered in Section 3.5). The transitive edges are double headed and the transitive activity pairs are double bounded. The $\geq 5$ that was attached to the rep node on the query is a shorthand for a sequence of 4 occurrences of the pattern followed by a regular rep node.

Intuitively, EX-patterns with or and rep nodes extend string regular expressions to concrete EX-pattern expressions. Namely, each EX-pattern defines a (possibly

\(^3\)An EX-trace without time stamps is defined as in Definitions 3.3.1 and 3.3.2 above, by dropping the time-stamp function $\tau$, and the corresponding constraints.
Figure 3.12: Execution pattern.

infinite) set of concrete EX-patterns, denoted \( \text{concrete}(p) \), as follows:

**Definition 3.3.4** Given an EX-pattern \( p \), \( \text{concrete}(p) \) is the set of all concrete EX-patterns that can be obtained from \( p \) by a sequence of the following replacement steps.

- **[or]** for some activity pair \((\text{act}(or), \text{com}(or))\), choose one of its internal traces and replace the subgraph consisting of the activity pair and all its internal traces by that chosen trace, connecting the incoming (outgoing) edges of \( \text{act}(or) \) (\( \text{com}(or) \)) to the start (end) node of the trace.

- **[rep, one occurrence]** for some activity pair \((\text{act}(rep), \text{com}(rep))\), replace the subgraph consisting of the activity pair and its internal traces by one copy of the internal traces, connecting the nodes previously pointing to (pointed by) \( \text{act}(rep) \) (\( \text{com}(rep) \)) directly to the start (end) nodes of the internal traces.

- **[rep, more occurrence]** for some activity pair \((\text{act}(rep), \text{com}(rep))\), plug a copy of the activity internal traces between \( \text{com}(rep) \) and its children nodes (with edges from \( \text{act}(rep) \) pointing to the start nodes of the traces and edges from the end nodes of the traces pointing to \( \text{act}(rep) \)'s previous children.)

To evaluate a query, the patterns in \( \text{concrete}(p) \) are matched to a given EX-trace. Such a match is called an *embedding.*
Definition 3.3.5 Let $p = (\hat{e}, T)$ be a concrete EX-pattern and let $e$ be an EX-trace. An embedding of $p$ into $e$ is a homomorphism $\psi$ from the nodes and edges in $p$ to nodes edges and paths in $e$ s.t.

1. **[nodes]** an activation (resp. completion) node is mapped to an activation (completion) node. Node labels are preserved; however, a node labeled by any can be mapped to any node.

2. **[edges]** each (transitive) edge from node $m$ to node $n$ in $p$ is mapped to an edge (path) from $\psi(m)$ to $\psi(n)$ in $e$. If the edge $[n, m]$ belongs to a direct internal trace of a transitive activity, the edge (s on the path) from $\psi(m)$ to $\psi(n)$ can be of any type (flow, or zoom-in) and otherwise must have the same type as $[n, m]$.

The start and end of $\psi$, denoted by $\text{start}(\psi)$ and $\text{end}(\psi)$, are the earliest and the latest time stamps of nodes in $\psi(p)$, respectively.

A pattern may have many matches in a given EX-trace. In some cases, users are satisfied by one match. In other cases, they may want to be informed on all (or some) matches. When one match suffices, it is desirable to find an early one. In the next section, we present an algorithm that is guaranteed to find a match, if one exists, and that can also find all matches, if so desired. The algorithm works in a greedy manner, matching pattern nodes to the earliest possible events. We next formally define the property of the first match it finds.

We use the following auxiliary notations. Given a concrete EX-pattern $p$, and an embedding $\psi'$ on a prefix $p'$ of $p$, we say that an embedding $\psi$ of $p$ extends $\psi'$, if $\psi$ agrees with $\psi'$ on $p'$. If $S$ is a set of embeddings of $p$, we denote by $S_{\downarrow \psi'}$ the set of embeddings in $S$ that extend $\psi'$, restricted to $\text{nodes}(p) \setminus \text{nodes}(p')$. When $S$ is a singleton $\{\psi\}$, we write $\psi_{\downarrow \psi'}$.

Definition 3.3.6 Let $p$ be an EX-pattern, $e$ an EX-trace, and $S$ a set of embeddings of patterns in concrete($p$) into $e$. An embedding $\psi \in S$ is greedy (in $S$) if the following holds:

(1) $\text{start}(\psi)$ is minimal in the set $\{\text{start}(\phi) \mid \phi \in S\}$. 
3.3. THE FORMAL MODEL

(2) Let \( n \) be the node with minimal time stamp in \( \psi \), i.e. \( \tau(\psi(n)) = \text{start}(\psi) \), and denote \( \psi \) restricted to \( n \) by \( \psi' \). Then, inductively, \( \psi_{1\psi'} \) is greedy in \( S_{1\psi'} \).

It is easy to show (by induction on the pattern size) that

**Proposition 3.3.7** For every EX-pattern \( p \) and every EX-trace \( e \), if the set \( S \) of embeddings of \( p \) into \( e \) is not empty, then an embedding that is greedy in \( S \) exists.

**Proof:** We show that if some embedding exists, a greedy one must exist as well. Given an embedding \( \psi \), define \( \text{greedy}(\psi, p) \) to be the set of nodes of \( p \) for which (*) above holds. If some node of \( p \) is not in \( \text{greedy}(\psi, p) \), let \( n \) be such a node with a minimal \( \tau(\psi(n)) \). There must exist some embedding that agrees with \( \psi \) on \( \text{greedy}(\psi, p) \), but maps some node \( n'' \) to a node with a smaller time-stamp. Let \( \psi' \) be such an embedding. Then we have that \( \text{greedy}(\psi, p) \cup \{ n \} \subseteq \text{greedy}(\psi', p) \). By induction, we obtain the existence of an embedding \( \hat{\psi} \), s.t. \( \text{greedy}(\hat{\psi}, p) \) contains all nodes of \( p \).

In some cases, the user may also be interested in consecutive occurrences of the same pattern, or some parts of it. For instance, in Example 3.2.1, once a corrupt auctioneer had been identified, we may want to be notified of further cancels performed by the auctioneer (i.e. we are interested in further occurrences of the sub-pattern inside the rep box).

Assume given an algorithm that given an EX-pattern \( p \) and an Ex-trace \( e \), finds an embedding \( \phi \) that is greedy w.r.t the set of all embeddings of \( p \) in \( e \). The following observation, that follows easily from the proposition, implies that the algorithm can easily be extended to find all embeddings of \( p \) into \( e \).

**Observation 3.3.8** If the set \( S \) of embeddings of \( p \) contains more than one embedding, and \( \psi \) is a greedy embedding in \( S \), then \( S \setminus \{ \psi \} \) is a non-empty set of embeddings, hence it contains an embedding \( \phi \) that is greedy in it, such that \( \text{start}(\psi) \leq \text{start}(\phi) \).

Note that since EX-patterns may contain choices (e.g. or) several greedy matches with the same start time may exist. Indeed, even for some concrete EX-patterns more than one greedy match exist, e.g. due to symmetry in the EX-pattern. If several such greedy matches exist, one can be chosen arbitrarily.
In the discussion above, we assumed that we are given the full EX-trace. But it clearly applies to prefixes of such traces as well.

### 3.4 Matching and optimization

We next explain how patterns matches are detected. We start by describing a simple pattern matching algorithm, then propose an effective optimization technique that exploits the BP specification to speed up computation, by focusing on the relevant parts of the events trace. It should be noted, however, that already the simple initial algorithm exploits knowledge about the common structure of BP traces, i.e. their nested DAG shape, to optimize the processing. In particular, when searching for an occurrence of a subpattern in the internal trace of a compound activity, if a completion event for the activity occurs, the algorithm immediately infers that the pattern can no longer occur in this internal trace and backtracks. (See details below). While this may remind the reader of XML stream-engines (which, when encountering an end-tag of an element, infer that the matching of a subpattern inside the element failed), there are two important differences which make the processing of BP patterns more intricate. First, BP patterns contain two navigation axes: the standard path-based navigation and the novel zoom-in navigation that allow users to query about activity flows that are nested at any depth inside the internal traces of compound activities. Second, unlike XML streams, where tree elements arrive in document order and siblings can be processed sequentially, in BPs the events of parallel sibling activities interleave. Here, a parallel processing of events according to their position in the flow is called for.

#### 3.4.1 Pattern Matching

We assume that the execution of processes, and matching of patterns, start at time 0. We are given an EX-pattern $p$ and our goal is to find the matches for $p$ in an (incrementally discovered) EX-trace $e$. At a time $t$, what is known from an EX-trace $e$ is only a prefix consisting of the nodes with time-stamp $\leq t$ and the edges between
them. Each arriving new event is appended to the prefix, with incoming edges of the two kinds described in Section 3.3.

To simplify the presentation, we first assume that \(p\) is a concrete EX-pattern. After presenting the algorithm for this restricted case, we explain how it extends naturally to general EX-patterns.

**Concrete patterns** The algorithm works in a greedy manner, trying to incrementally extend a greedy embedding for a prefix of \(p\) (initially empty), to a greedy embedding for a larger prefix. On failure it backtracks, refines the prefix embedding and retries to proceed again. Given an EX-pattern \(p\) we construct an automaton \(A\) whose states are the nodes of \(p\). Its start (resp. end) states correspond to the start (end) nodes of \(p\). A state can be active or inactive. Initially, only the start state is active. Other states become active once they get activation messages from all their respective parents, or due to the \texttt{backtrack} operator described below.

We maintain two data structures for backtracking. The first, an (initially empty) list called the \texttt{events-list}, contains trace nodes that may need to be (re)processed. Each new event (node) is appended to its end. The second, called \texttt{tested}, is a map from a subset of the states to events in \texttt{events-list}, representing the embedding computed thus far for some prefix of the pattern. Initially the mapping of all states is set to null. Each state (pattern node) maintains a \texttt{current-event} variable that points to an event in the \texttt{events-list} that the state needs to process. If it points to the place after last in the list it means that the state awaits the arrival of a new event. Initially the \texttt{current-event} of the start state points to the beginning of \texttt{events-list} and the \texttt{current-event} of all other states are set to null.

Each active state executes the algorithm depicted in Fig. 3.13. We assume that every iteration of the algorithm (the body of the while loop), which involves reading and possibly writing in the data structures and (in)activating some states, is executed atomically (our implementation uses for that a simple locking mechanism.)

Each active state \(s\) reads iteratively events from \texttt{events-list} (line 2) and processes them. This processing stops when \(s\) becomes inactive, and restarts when \(s\) becomes active again. If an event matches the conditions on the state and on its
Automaton state $s$

While $s$ is active do:

1. $n = \text{current-event}$.
2. Advance current-event to next event in events-list.
3. If $\text{match}? (s, n)$
   
   (a) Set $s$'s entry in tested to point to $n$.
   
   (b) Inactivate $s$.
   
   (c) Send an activation message to the children states of $s$, setting their current-event to that of $s$.

4. Else % not matched %

5. If $n$ is a completion event, $s$ is a completion state, and the activation event of $n$'s activity is assigned in tested to the activation counterpart of $s$,

   backtrack($s$)

6. Else, if $n$ is a completion event for one of $s$'s ancestors in the zoom-in hierarchy, or a completion event for the end activity,

   backtrack'($s$)

End While

Figure 3.13: Processing events.

incoming edges (line 4), it is added to tested (line 5). The state is then inactivated (line 6) and we proceed with (i.e. sends activation message to) its children⁴ (line 7). The $\text{match}?$ predicate is depicted in Figure 3.14. It tests whether, given a state $s$, an event $n$, and the tested entries of the parents of $s$, $n$ is a potential assignment for $s$.

If a match of an activation node fails, the event is skipped, and the algorithm proceeds to the next event. For a failure of a completion event, we consider two cases: First (line 9), if the state represents activity completion whose activation counterpart was matched in tested with the activation counterpart of the given event (but the given state and event nevertheless don’t match), this implies a failure for the part of the pattern involving the implementation of the activation/completion pair. The algorithm then backtracks (line 10), trying to find another match for this part. The backtrack operator is described in Figure 3.15. It finds the last point where a decision was made on the matching of a previous relevant activity $\hat{s}$ and retries from

---

⁴A child becomes active after receiving messages from all its parents. It then starts reading events from the last current-event it received.
3.4. MATCHING AND OPTIMIZATION

match?(state s, event n): boolean

1. If (a) n’s labels satisfies the label conditions of s, and
2. (b) for every parent \( \hat{s} \) of s, tested contains some assignment
   \( \hat{n} \) to \( \hat{s} \) and the trace path from \( \hat{n} \) to n satisfies the conditions
   on the edge \( (\hat{s}, s) \) in the pattern,
3. return True
4. Else return False

Figure 3.14: Matching an event.

backtrack(state s)

1. Choose an ancestor \( \hat{s} \) of s, whose event in tested is an activation
   event with maximal timestamp (among s’s ancestors).
2. Clear, in tested, the entries of \( \hat{s} \) and its descendant states.
3. Inactivate the descendants of \( \hat{s} \) and set their current-event to null,
4. Reactivate \( \hat{s} \).

Figure 3.15: Backtrack.

there. A second type of failure that needs treatment (line 11) is when the event is
a completion for an activity whose activation is assigned in tested to an ancestor \( s' \)
of s (or similarly, if the event is the last event of the trace). This implies a failure
to match the part of the pattern from \( s' \) to its completion (resp. from the beginning
of the pattern to its completion). Here, the algorithm backtracks (lines 12), trying
to find another match for this part, using a different version of backtrack, denoted
backtrack’, that considers for backtracking only ancestors in the same or higher level
in the zoom-in hierarchy.

A match for the pattern is identified if&when the final state of the automaton is
inserted into tested. On success, tested contains the match for the pattern nodes.
The mapping for the edges consists of the edges/paths that were used to qualify
these assignments in line (2) of the match? procedure. To find consecutive matches,
if requested, a backtrack operation is applied to the final state and the matching
process continues to find the next match. The matching fails if all active states read
the end event of the trace before a successful match is found. An earlier detection of
failure is considered below.

We now demonstrate the algorithm.

---

5The reactivated \( \hat{s} \) now begins to read events starting from its current-event.
Example 3.4.1 Figure 3.16 illustrates an EX-pattern and an EX-trace. The nodes of the EX-pattern, which comprise the automaton constructed by the algorithm, are marked $s_1, \ldots, s_{10}$, and the nodes of the execution trace are marked $e_1, \ldots, e_{18}$. Figure 3.17 demonstrates the execution steps of the algorithm on these events. The columns show the active states (at the beginning of each step), the current-event, the tested list (after the event was processed), and the decision taken: found a match, skip the event, or backtrack. The algorithm matches the activation event of $A$ ($e_1$) to $s_1$, and the completion event of $A$ ($e_2$) to $s_2$. Then it tries to match $e_3$ with $s_3$, skips on $e_4$, matches $e_5$, $e_6$ to $s_9$, $s_{10}$, skips $e_7$, and then backtracks $s_{10}$ when failing to match $e_8$ (conditions on only one of the incoming edges are satisfied). Backtracking $s_{10}$ is not enough, therefore the algorithm backtracks the first wrong decision, $s_3$, and then continues. Another backtracking occurs when $e_{13}$ fails to satisfy the condition on the non transitive edge from $S_6$ to $S_7$, and finally the algorithm ends with success.

Discussion To understand why the algorithm indeed computes a greedy match, let’s look at the following. A first observation is, that if there exist some embedding, the algorithm finds it. This is inferred from the fact that in the worst case, when the
3.4 MATCHING AND OPTIMIZATION

Figure 3.17: Algorithm execution example.
algorithm encounters the end of the trace before finding an embedding, it performs an exhaustive search by repeatedly backtracking the automaton states up to the initial state, trying to match the trace of events in each point.

Second, the algorithm works in a greedy manner, trying to incrementally extend a greedy embedding for a prefix of the pattern, to a greedy embedding for a larger prefix. If there are no failures that require backtracking, if the algorithm finds an embedding it is a greedy one. On failure the algorithm backtracks, to the state whose matching event is an activation event with maximal timestamp. Then, the algorithm continues to process the events from this point. Additional backtracks are handled the same way. We can see that when the algorithm backtracks, it tries matching all next events according to their timestamp order, guaranteeing that if it finds an embedding, for each prefix it will have the smallest timestamp.

The worse case time complexity of the algorithm is polynomial in the size of the trace (with the exponent determined by the size of the pattern). The intuition is that the algorithm exhaustively checks all relevant embeddings (as described above), and the upper bound on their number is polynomial in the size of the trace (with the exponent determined by the pattern size).

Before extending the algorithm to work with general patterns, let us comment about some of its properties.

**Remark 1** The algorithm works greedily, in a deterministic manner, attempting to match events as early as possibly and backtracking on failure. Two possible alternatives could be (1) to use a non deterministic automaton that checks simultaneously all possible embeddings, thus avoiding backtracking, and (2) to construct some deterministic variant of that non deterministic automaton. Just like for standard regular expressions, a disadvantage of the first approach is the need to manage simultaneously a large number of active states [75]. A disadvantage of the second approach is the potential exponential growth in the size of the automaton [74]. Our algorithm provides a hybrid solution. We use a small automaton with the same size as the pattern, and since states are inactivated as soon as a matching event is assigned to them, only relatively few states are simultaneously active. The price paid for this is
3.4 MATCHING AND OPTIMIZATION

the need for backtracking. An optimization technique that allows to identify failures early and thus to avoid some redundant work and backtracking is presented below. Our experiments, presented in Section 3.6, show the optimized algorithm to be extremely efficient.

Remark 2 The events of the trace are recorded in events-list for backtracking. It is easy to see that an event \( n \) will never be reprocessed if \( n \) and its preceding events are not pointed by tested or any of the current-event variables of the states. Such an event can be removed from the list. We show below that the optimization technique mentioned above is also useful for identifying such redundant events.

It is possible to build (rather artificial) scenarios where all events must be retained in events-list “for ever”. For example, consider a BP with an activity \( A \) that invokes itself (recursively) and may also, arbitrarily later, invoke some other activity \( B \). Assume that our query searches for an \( A \) activity that invoked both \( A \) and \( B \). If the given BP trace contains a long sequence of \( A \)’s, we need to keep them all since we do not know in advance which of them (if any) will invoke a \( B \) later on. The problem here is that all the \( A \) activities remain ”alive” for an unbounded time, hence
may invoke new children activities arbitrarily late. In practice, in a typical BP, the number of individual activities that are kept alive unboundedly is bounded, so such phenomena are unlikely to occur. Indeed, in all the real life examples we examined, the number of events that needed to be retained was fairly small and proportional to the pattern size.

**Handling or and rep** We now describe the adjustments needed to handle or and rep.

**[or]** Consider an activity pair \((\text{act}(or), \text{com}(or))\) in \(p\). When the automaton state \(s\) of \(\text{act}(or)\) (resp. the state \(s'\) of \(\text{com}(or)\)) is activated it does not read any events but immediately sends activation messages to all its children (with its current event). For \(s'\) to get activated it suffices that it receives an activation message from one of the activity internal traces. The children of \(s\) (resp. \(s'\)) check \text{match}\? w.r.t their grandparents rather than their parents (or great grandparents if the grandparents are also or nodes). For the children of \(s'\), a more lenient version of \text{match}\? is employed, where condition (b) needs to be satisfied only for the grandparent that activated \(s'\).

Since \(s'\) may now be activated several times, due to several branches of the or, and consequently its children may be matched to several events, we maintain
### 3.4. MATCHING AND OPTIMIZATION

#### Example 3.4.2

This example demonstrates how the algorithm handles *or*. The pattern and a trace are shown in Figure 3.19, and the algorithm execution steps are described in Figure 3.20. The algorithm matches $s_1, s_2$, then it activates $s_3$ that immediately activated $s_6, s_8$, and $s_{11}$. It matches $e_4, e_5, e_6$ to $s_6, s_4, s_7$ resp. (tested items of each branch are underlined), then $s_8$ is activated, since it suffices for an or state to get one activation message. This deactivates all active or branches (i.e.,

<table>
<thead>
<tr>
<th>active</th>
<th>$e_i$</th>
<th>tested</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>$e_1$</td>
<td>$s_1 \rightarrow e_1$</td>
<td>✓</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$e_2$</td>
<td>$s_1 \rightarrow e_1, s_2 \rightarrow e_2$</td>
<td>✓</td>
</tr>
<tr>
<td>$s_3$</td>
<td>–</td>
<td>$s_1 \rightarrow e_1, s_2 \rightarrow e_2$</td>
<td>✓</td>
</tr>
<tr>
<td>$s_4, s_6, s_{11}$</td>
<td>$e_3$</td>
<td>$s_1 \rightarrow e_1, s_2 \rightarrow e_2$</td>
<td>skip</td>
</tr>
<tr>
<td>$s_4, s_6, s_{11}$</td>
<td>$e_4$</td>
<td>$s_1 \rightarrow e_1, s_2 \rightarrow e_2, s_6 \rightarrow e_4$</td>
<td>✓</td>
</tr>
<tr>
<td>$s_4, s_7, s_{11}$</td>
<td>$e_5$</td>
<td>$s_1 \rightarrow e_1, s_2 \rightarrow e_2, s_6 \rightarrow e_4, s_4 \rightarrow e_5$</td>
<td>✓</td>
</tr>
<tr>
<td>$s_5, s_7, s_{11}$</td>
<td>$e_6$</td>
<td>$s_1 \rightarrow e_1, s_2 \rightarrow e_2, s_6 \rightarrow e_4, s_7 \rightarrow e_6, s_4 \rightarrow e_5$</td>
<td>✓</td>
</tr>
<tr>
<td>$s_8$</td>
<td>$e_9$</td>
<td>$s_1 \rightarrow e_1, s_2 \rightarrow e_2, s_6 \rightarrow e_4, s_7 \rightarrow e_6$</td>
<td>✓</td>
</tr>
<tr>
<td>$s_9$</td>
<td>$e_{10}$</td>
<td>$s_1 \rightarrow e_1, s_2 \rightarrow e_2, s_6 \rightarrow e_4, s_7 \rightarrow e_6$</td>
<td>skip</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>$s_1 \rightarrow e_1, s_2 \rightarrow e_2, s_6 \rightarrow e_4, s_7 \rightarrow e_6$</td>
<td>...</td>
</tr>
<tr>
<td>$s_{10}$</td>
<td>$e_{10}$</td>
<td>$s_1 \rightarrow e_1, s_2 \rightarrow e_2, s_6 \rightarrow e_4, s_7 \rightarrow e_6$</td>
<td>backtrack($s_9$)</td>
</tr>
</tbody>
</table>

| $s_5, (s_6), s_{11}$ | $e_7$ | $s_1 \rightarrow e_1, s_2 \rightarrow e_2, s_4 \rightarrow e_5, s_5 \rightarrow e_7$ | ✓      |
| $s_8$  | – | $s_1 \rightarrow e_1, s_2 \rightarrow e_2, s_4 \rightarrow e_5, s_5 \rightarrow e_7$ | ✓      |
| $s_9$  | $e_7$ | $s_1 \rightarrow e_1, s_2 \rightarrow e_2, s_4 \rightarrow e_5, s_5 \rightarrow e_7$ | ✓      |
| $s_{10}$ | $e_9$ | $s_1 \rightarrow e_1, s_2 \rightarrow e_2, s_4 \rightarrow e_5, s_5 \rightarrow e_7, s_9 \rightarrow e_9, s_{10} \rightarrow e_{10}$ | success |

Figure 3.20: *or* execution example.

in **tested** a set of events for each state, corresponding to the various possible matches. The context of each matching (i.e., to which choices of *or* branches it corresponds) is recorded with the events, and all consequent tests/operations take into consideration only assignments relevant to the given context.

[rep] The processing of *rep* follows similar lines. It is based on the observation that an activity pair ($\text{act}(\text{rep}), \text{com}(\text{rep})$) in $p$, which stands for one or more repetitions of some subpattern $p'$, can be viewed as an *or* between the pattern $p'$ and the pattern containing one occurrence of $p'$ followed by another *rep* of $p'$. This “virtual” *or* is treated as above, recursively.

Here, we slightly modify the backtrack($\hat{s}$) function to skip line 2 in case $\hat{s}$ that was chosen in line 1 is $\text{com}(\text{rep})$. So, we backtrack without clearing the embedding of the nodes inside the $\text{com}(\text{rep})$. 

---

---
$s_5$, $s_{11}$), and clears their matches from $tested$ (while saving them aside for backtracking). At the next step $s_9$ is activated, but the algorithm fails to match it, and backtracks when the stream of events ends. The backtrack operation activates again the $or$’s branches, while backtracking the latest match $s_7$ and reactivating $s_6$ with its next event. The other branches are reactivated, each with its $current-event$ and $tested$. Then $e_7$ is matched to $s_5$ and so on, until the algorithm ends with success.

**Example 3.4.3** This example demonstrates how the algorithm handles $rep$. The pattern and a trace are shown in Figure 3.21, and the algorithm execution steps are described in Figure 3.22. The execution steps are trivial until matching of $s_5$ activates $s_6$ which is a completion of $rep$. Now, similarly to the behavior of $or$ both $s_7$, the next state, and $s_3$, the activation of the $rep$, are activated. Both states continue in parallel, one tries to go forward, and the other tries to match another repetition. The algorithm ends with success when the whole pattern is matched.
3.4. MATCHING AND OPTIMIZATION

3.4.2 Optimization

So far, our algorithm ignored the BPEL specifications of the monitored processes. Let us now see how to use them to avoid redundant processing and to record only useful history.

As a simple example, consider the query in Figure 3.7, that monitors corrupt sellers, and the auctionHouse BP in Figure 3.1. If the process trace reports an invocation of the buyer process compound activity, we immediately know that this activity, as well as all the events in its internal trace, are irrelevant to the query pattern and can be ignored. Furthermore, since the BP specification indicates that only one of buyer process and seller process can occur in a given process instance, we can infer that the invocation of buyer process is inconsistent with the pattern and a match for the pattern is impossible. These notions are now formally defined.

**Definition 3.4.4** Let $S$ be a BPEL specification and $o$ an activity in $S$. Given an EX-pattern $p$ and a node $n$ (resp. an edge $e$) in $p$, we say that the activation/completion of $o$ is irrelevant to $n$ (resp. $e$) if there is no embedding of $p$ into an EX-trace of $S$ where $o$’s activation/completion event is assigned to $n$ (resp. appears on a path assigned to $e$).

We say that $o$ is inconsistent with $p$ if $p$ cannot be embedded into any EX-trace of $S$ that contains an activation event of $o$.

We explain below how irrelevancy and inconsistency are determined. For now, assuming that such a detection algorithm is given, we show how it can be used to
refine the algorithm described above.

- When an active state reads events from the events-list, it can ignore the events that are irrelevant to the corresponding pattern node. This prevents event assignments that will for sure be detected later as unfit. Events that are irrelevant to all the pattern nodes and edges need not be recorded in the events-list. An event \( n \) that is relevant to some of the states/edges may be removed from the events-list as soon as it is not pointed by tested or any of the current-event variables of the states, and is also irrelevant to all states with current-event pointing to a preceding event and their descendent states and edges (as it will never be useful for backtracking).

- When an active state reads an activation event for an activity that is inconsistent with the query pattern it can immediately declare failure and stop the query processing.

- Similarly, if an active state reads an activation event for an activity is inconsistent with the internal trace of some of the state’s ancestors through the zoom-in relationship (w.r.t the specification of the activity currently assigned to that ancestor), a backtrack operation to the ancestor can be issued. This early backtracking eliminates future redundant matchings.

**Testing irrelevance and inconsistency** We now explain the main ideas of our algorithm for testing irrelevance and inconsistency, this is a work in progress. To check the irrelevance of the activation/completion of an activity \( o \) to a node \( n \) (edge \( e \)) in the pattern \( p \), one needs to test if an instance of the given BP may contain a subtrace of shape similar to \( p \) where the activation/completion of \( o \) represents \( n \) (resp. appears on the path that represents \( e \)). If not, the activity is irrelevant to the pattern node (edge). To check inconsistency one needs to check if a BP instance that contains both an activation of \( o \) and a subtrace of the shape \( p \) may exist. Again, if not, the activity is inconsistent with the pattern.

Analyzing the possible runs of a BP is essentially a verification problem [70] and is typically of very high complexity (from NP-hard for very simple specifications to
3.4. MATCHING AND OPTIMIZATION

Figure 3.23: Irrelevance and inconsistency.

undecidable in the general case [107]). In the previous chapter we presented BP-QL, a query language for analyzing BP specifications, with polynomial data complexity and a graphical interface. Having syntax close to that of BP-Mon, it is easy to transform BP-Mon patterns into BP-QL queries on the specification, that capture irrelevance and inconsistency.

Note that to guaranty low complexity, BP-QL ignores the run-time semantics of certain BPEL constructs such as or, conditional execution and variable values. Hence, these queries capture irrelevance and inconsistency in a safe manner. Namely, a query may miss some cases of these properties, but those that are identified as such are indeed irrelevant/inconsistent. Optimization-wise this only means that some optimization opportunities may be missed, but the correctness of the matching algorithm is not compromised.

Example 3.4.5 The query in Figure 3.7, monitors sellers actions to guarantee fair play. If we look into the spec (see Figure 3.1), we can see that after a user chooses to play a buyer role, the execution trace is irrelevant to the monitoring process and all further events can be ignored. By transforming the monitoring query to a BP-QL query
on the spec (see Figure 3.23), we can retrieve the activities that may be relevant to the monitoring query, and direct the process to filter out all other irrelevant events.

**Example 3.4.6** Checking inconsistency may be useful for early backtracking. One example is early backtracking of wrong decision in case of non transitive edges (e.g. the query in Figure 3.8). Recall that in the matching algorithm above, there is no special case to handle failures caused by non transitive edges (where a matched event should immediately be followed by a certain event). Such failures are backtracked when an ancestor match is backtracked or when the trace ends, resulting with some redundant computing. A nice optimization can be achieved by checking inconsistency of match candidates for nodes which have one or more non transitive outgoing edges.

It is important to note that since we are querying the BP specifications, all the decisions regarding the potential inconsistency (irrelevance) of activities with (to) the pattern (pattern nodes) can be made statically, at compile time, before the monitoring starts, hence cause no delays in the actual monitoring processes.

### 3.5 The full language

For simplicity, we used so far a very simple data model and ignored report generation. We now briefly consider useful extensions that enhance the expressive power, and facilitate the monitoring of real life business processes.

**Data values and predicates** In practice, an execution trace carries additional information about the performed activities, such as the names and values of data variables. This can be modeled by labeling the nodes in both EX-traces and EX-patterns with this additional data, requiring the embedding to respect these labels as well. Additionally, EX-patterns may also use label or path predicated. For instance, rather than searching specifically for `cancel_auction_request` and `cancel_bid_request` activities, one may ask for all the activities whose name contains the string “cancel”. One can also use predicates on the activity time-stamps to focus the search on certain time intervals. In an embedding, a pattern node/edge labeled by a predicate must be mapped to trace node/path whose label satisfies the predicate.
3.5. **THE FULL LANGUAGE**

**Variables and joins** Ex-patterns can be extended by attaching variables to concrete activities and edges, and by (in)equality conditions on variables. Then, pattern activities attached to (un)equal variables are mapped to trace activities all having (distinct)identical labels, and pattern edges labeled by (un)equal variables are mapped to paths whose sequences of labels are all (different) equal words. When a variable is attached to an activity (edge) in the scope of a rep, we require that, in the embedding of the corresponding concrete pattern, all the occurrences of the activity (edge) of the variable satisfy the (in) equality conditions on it.

**Querying specifications** In some cases we want to relate monitoring to specific activity patterns that appear (or not) in the BP specification. See, e.g., Example 3.2.4 in Section 3.2. To query specifications, we rely again on BP-QL. Queries then consist of two parts, that query the specification and the execution trace, respectively. (See, e.g. Figure 3.11). Join conditions between variables attached to the nodes/paths of the two parts provide the glue between them.

**Distributed systems and queries** In a distributed setting, each peer holds a set of BPs and may provide (resp. use) activities to (of) remote peers. Users may wish to monitor these remote components as well, provided access is allowed by the respective organizations. The data model and query language extend naturally to this setting, associating peer ids with activity pairs in execution traces/patterns. When an activity pair $s, s'$ in a query is annotated by a peer id $P$, the search for its internal EX-pattern is restricted to traces supplied by $P$. More generally, queries may use predicates on peer ids to restrict the search to a specific family of peers.

The pattern matching algorithm presented in Section 3.4 extends naturally to this distributed setting. To avoid shipping events between sites, the (sub)automaton $\hat{A}$, corresponding to the internal EX-pattern of an activity pair $s, s'$ annotated by $P$ is installed on the peer $P$. When $s$ is matched, it notifies the start node of $\hat{A}$; a matching for $\hat{A}$ is computed on $P$ (as described in Section 3.4). On success, $s'$ is informed (and is activated). If the matching fails, $s$ in notified (and consequently backtracks).
CHAPTER 3. MONITORING BUSINESS PROCESSES WITH BP-MON

Figure 3.24: XML document.

Reports We conclude by considering report generation. Assume first a report attached to the end node of an EX-pattern $p$.

A match $(p_c, \psi)$ for a concrete EX-pattern $p_c \in \text{concrete}(p)$ can be viewed as an XML document (tree), that records the $\psi$ assignment for the activities (activation-completion pairs) and the edges in $p_c$. Each match found by the algorithm of Section 3.4 generates one such XML entry. When consecutive matches for a (sub)pattern are requested, they also generate each one such entry. For example Figure 3.24 demonstrates the XML stream created by the Query in Figure 3.9. This query retrieves all repetitions of the pattern. As you can see, there is an element $\text{rep}$ for each match, and below it, the hierarchy of activities that lead to the selected activities (denoted as $x$ and $y$). The output includes the operation name, the activation and completion timestamps, and activity data (when applicable) for each activity. The $\text{Report}$ command is applied to this stream of matches. The syntax and semantics resembles that of previous proposals for such reports [106, 36]; we explain here the main constructs. By default a report is issued for each entry. To issue a report only when certain conditions are satisfied a $\text{When cond}$ statement can be used, where $\text{cond}$ is a boolean condition on the value of attributes or aggregate functions (described below). Periodic reports may be generated by the $\text{Every time}$ command, where $\text{time}$
may be a time interval or the number of entries generated since last report. A sliding window describing the entries relevant for the generation of the report can be defined using the \textit{Range time} command. The structure of the report - an XML document - is described in a manner similar to that of the return clause of XQuery and may include grouping of entries and aggregations like average, max, min, count, sum. Two reporting modes are available: A \textit{local} report is issued for a given process \textit{instance} and uses only entries of that instance. A \textit{global} report is issued per BP and uses entries of all the BP instances.

In general, report commands may be attached to any node in the pattern. The portion of the execution pattern relevant to such a report consists of the prefix of the pattern including the report node and its predecessors. The report generated for such a (sub) pattern ignores the rest of the pattern, and is processed in the same way as described above.

\textbf{Remark:} A naive, and very inefficient, approach to process a query with several report nodes is to compute, separately, the first matches of each of their respective pattern prefixes. Recall however that our algorithm works in a greedy manner by matching pattern prefixes, then expanding them to matches for larger prefixes. This can naturally be exploited to factorize the common processing, computing first matches for reports of “shorter” prefixes and then expanding them, when possible, to the reports of larger prefixes.

\section{3.6 Implementation and Experiments}

\subsection{3.6.1 Implementation}

The query language and algorithms presented above have been implemented and tested in the \texttt{BP-Mon} monitoring system. To support flexible deployment, the system compiles \texttt{BP-Mon} queries into BPEL specifications. The specification $S(p)$ generated for a query pattern $p$ describes a process (essentially the automaton described in the previous section) that will perform the monitoring task for $p$. $S(p)$ is then automatically compiled into an executable code to be run on the same BPEL application.
The system architecture is depicted in Figure 3.25. We describe below the various components.

**Visual editor**  
**BP-Mon** queries are written via a visual editor, in one of two modes. The user can draw the patterns from scratch, using a drag-and-drop items palette. Or, starting from a specification of a BP $p$, use a wizard to create queries to monitor $p$, as follows: The user marks the nodes of $p$ that she wishes to include in the query. Then by one click a query draft is created, where non selected nodes are omitted and the selected nodes are connected with transitive edges that reflect their flow and zoom-in relationship in $p$. The user can then add conditions on node values, add report points, make some final adjustment, and click a button to deploy the query on a BPEL server. The visual interface is implemented as an Eclipse [61] plug-in, similarly to Oracle BPEL designer, allowing both products to run simultaneously in the same framework.

As an example, Figure 3.26 shows (part of) the BPEL specification of the auctioning BP with the nodes selected (in this part) by the user. The generated monitoring query, after some user adjustments, is shown in Figure 3.27, ready to be deployed. The system provides two alternatives for defining a report. The first (see Figure 3.28(a)), is using a simple form to define the **BP-Mon** report, that will be automatically translated to XQuery. This option allows only a subset of XQuery. The second option
3.6. IMPLEMENTATION AND EXPERIMENTS

(Figure 3.28(b)) is writing an XQuery query directly. The pre-prepared form in allows to choose reporting on first match report, or a report* that searches and reports on consecutive occurrences of the searched pattern, define a boolean condition when on the value of attributes or aggregate functions, choose the frequency that the report will be issued, and the range for the sliding window. Then the user defines the output structure using a syntax similar to the return clause of XQuery on the XML documented that is generated by the query, using an editor that supports auto-completion, and popups a list of the available XML elements and attributes whenever a ” is typed. Finally, the user can add a group by having clause to the query. Figure 3.28(b), shows the equivalent query, using a free format editor. On the left there is a tree-view display of the XML document, and on the write an editor that supports auto-completion and drug-and-drop of elements. This example demonstrates how the predefined clauses in the query form, including the group-by-having clause, are translated to XQuery format.

Query translator The query translator compiles a query on \( p \) to a BPEL process - the Query Process (QP for short) in Fig. 3.25 – that implements the automaton described in Section 3.4. Each state is implemented as a compound activity consisting
CHAPTER 3. MONITORING BUSINESS PROCESSES WITH BP-MON

Figure 3.27: Edit query.

Figure 3.28: (a) Report form. (b) XQuery.
of two components, one in charge of reading the incoming events, the other in charge of events processing and backtracking. The QP is deployed onto the BPEL server where the instances of \( p \) are executed. Several QPs, monitoring the same or different processes, may be deployed on a server. Note that in principle one may even have queries that monitor the execution of other queries!

A Java library, shared by all the QPs, handles the management of the events queue and the related data structures.

**Dispatcher**  The *dispatcher* module is responsible for the run-time mapping between the events of BP instances and the QPs. It subscribes to relevant events of the queried BPs when a query is deployed, and receives the relevant events generated by instances of these BPs (as described in Section 3.2). The first event from a new BP instance causes the dispatcher to create a new instance of relevant QPs. Further events are delegated to the running QP instances.

**Report generation**  The final step is generating the reports. As explained in Section 3.5, a successful matching for the query pattern associated with a report node generates an XML entry recording the embedding, and the *Report* command is applied to this stream of matches. Observe that from this point and on, since all the special BPEL-related issues that we mentioned in Section 3.1 have already been treated by the BP-Mon engine, we are back to standard XML stream processing, and can use a standard such engine to generate the report. In our implementation we support two alternatives for report generation. The first uses the streaming system of [106]. This streaming system is actually relational, but the fairly simple structure of the BP-Mon XML allows for natural translation to relational format and back to XML. The second uses a lightweight in house reporting tool based on XQuery and XSLT. But in principle any XML streaming tool that supports the needed reporting features can be plugged into our architecture.
3.6.2 Experiments

The implementation by translation of queries into BPEL processes, then running them on the same server as the queried processes, has two main advantages: Portability of queries between BPEL engines; and a great simplification of the software development, exploiting the infrastructure provided by such engines for parallel and distributed process management, and software composition. The price paid for this is the extra load on the BP server who now needs to also run query instances. To estimate the overhead incurred by running the query on the same server, the performance impact on the queried processes, the scalability of the solution, and the effectiveness of the optimizations, we ran several experiments.

We considered BPs with varying number of activities, where the monitoring involves different percentage of the activities in the BPs. We varied the ratio of processes vs. queries, and also varied the type of the monitored processes, from I/O bounded BPs, to CPU bounded ones.\textsuperscript{6} Since the generation of reports is fairly standard, we focused on the parts specific to BP-Mon, namely the matching of patterns; our measurements do not include report generation time.

In the experiments, we used a family of processes consisting of sequences of nested \texttt{while} constructs, with atomic activities that each invokes a given Java class, some run in parallel, and with an optional \texttt{wait} activity between them. By configuring the number of \texttt{while} iterations and the properties of the Java class we could vary the size of the process, the characteristics of the activities (I/O or CPU bounded), and the percentage of queried process activities (our queries queried activities appearing only in some of the loops). The queries use the Report* option that requires matching of consecutive occurrences of the searched pattern in the EX-trace (as this requires more processing than a single Report).

In the experiments, we measured execution time (in seconds) of processes and queries. The tests were performed on Pentium4 3.0GHz, dual core with 1GB RAM memory, running Windows XP Professional, JBoss AS 4.0.4. Oracle BPEL Process Manager 10.1.2. with Oracle 9i database. A representative sample of results is shown

\footnote{\textsuperscript{6}Since our experiments showed no significant difference between I/O bound and CPU bound BPs, the results discussed below use activities with a uniform mix of I/O and CPU load.}
3.6. IMPLEMENTATION AND EXPERIMENTS

Figure 3.29: Queries overhead.

Figure 3.30: Varying number of queried processes.
in Figures 3.29 - 3.32.

Figure 3.29 demonstrates the very minimal overhead of our solution as well as its scalability. Each BP here consists of 200 activities; the monitored patterns involve about 40% of the activities. The graph shows, for a varying number of BPs, four measurements of total execution time. The first (left-most) column in each set shows the execution time of the BPs, with event generation, but without monitoring. The second column shows the execution time of the BPs when monitored, with one query per process. Clearly, the overhead on process execution due to monitoring is very low. The third column shows the execution time of the queries. As should be expected, their execution time is slightly higher than the processes themselves – a query is invoked with the process, but lags behind a bit when processing its events. (Recall that the queries here report consecutive occurrences of the searched pattern in the EX-trace, hence continue the monitoring till the process ends. Queries that report just one occurrence stop as soon at it is detected and thus entail even lesser overhead). Obviously, all the results are affected by the scalability of the BPEL server itself. We can see that the execution time grows linearly with the number of concurrent processes.

The queries that we show here have 3 reporting points. Recall that one of our optimizations is factorizing the common pattern matchings for the reports. To illustrate the reduction in processing time that this achieves, the forth column shows what would be execution time for the three matches if computed separately.

In the above experiment all process instances are monitored, each by one query. To measure the effect of changing these parameters, we varied the overall number of queries, assigning to each process a subset of random size, with uniform distribution.
3.6. IMPLEMENTATION AND EXPERIMENTS

Figure 3.32: Impact of Optimization.

Figure 3.30 illustrates representative results, for 50 process instances with parameters the same as above (200 activities, of which 40% occur in the monitored patterns), and the average number of queries per process varying from 0 to 2. We can see that the growing number of queries has only minimal effect on execution time. Indeed as already seen in the previous experiment, the execution time is mostly affected by the running time of the monitored processes and the overhead due to query processing is marginal.

Figure 3.31 illustrates the effect of monitoring different percentage of the activities in a process. We ran the experiment with the same 50 process instances as above, and query patterns involving 10% to 100% of the BP activities. The execution time grows moderately with the percentage of monitored activities. In practice the common case is likely to be close to the lower left part of the curve, as typical BP specifications are large with only small part being relevant for a particular monitoring task.

We conclude by considering the effect of our optimization technique of pruning redundant monitoring based on an analysis of the BPEL process specification. Figure 3.32 illustrates the improvement achieved by applying this method. The scenario here is similar to what we have seen in Example 3.2.1: the BPEL specification has a switch construct, and only one branch is relevant to the query. The process instances choose randomly one of the branches. We measured the execution time of optimized and non-optimized queries, varying the number of process instances. The experiments shows a performance gain of almost 50%. This is not surprising, since about 50% of the processing, that involves non interesting branches, was avoided.
Of course, performance improvement in general will depend on the mix of processes, queries, and their properties.

3.7 Summary

We presented BP-Mon, a novel query language for monitoring BPs. BP-Mon offers a high level intuitive design of monitoring tasks. The language is visual and intuitive with tight analogy to the spec, and supports flexible description of execution patterns (sequential and parallel executions, and/or, repetitions). The language facilitate several means of reporting, XML reports, notifications and invocation of corrective actions.

We presented a greedy algorithm that is based on matching pattern prefixes, then expanding them to matches for larger prefixes. We also presented a novel optimization technique that exploits available knowledge on the BP structure to speed up computation by pruning of redundant monitoring based on irrelevance and inconsistency.

BP-Mon allows to design complex monitoring tasks that deal with both events and flow. It offers easy, user-friendly design of such tasks; and it compiles these tasks into standard BPEL processes, thus providing easy deployment, portability, and minimal overhead.
Chapter 4

Integration & Customization of Web Applications

4.1 Introduction

In this chapter 4, we consider the problem of integrating and customizing existing http-based applications. This part of the thesis took place a few years ago, before Web Services standards emerged. The various standard proposals at that time gave no solutions to this problem. The proposed framework supports a declarative specification language for specifying the integration and customization task, covering the full profile of e-commerce applications. Then, acting as an application generator, the system generates a full integrated/customized e-commerce application. Indeed, the BPEL standard that appeared three years later, takes a similar approach to our work, and provides a viable solution to this problem. Similarly to our framework, the BPEL standard uses a declarative specification of the process to automatically generate the code that implements it. However, there are still aspects like integration of heterogenous data, which are not part of the standard. The results of this work are still relevant, and can easily be adapted to use the BPEL standard as we show in Section 4.6.
Electronic commerce applications support the interaction between different parties participating in a commerce transaction via the network, as well as the management of the data involved in the process [138]. The Internet provides access to a large and diverse body of such applications, e.g. book and record stores, supermarkets, train and airline reservation systems, etc. The variety of available applications makes electronic shopping appealing for both customers and entrepreneurs. From the customer’s viewpoint, the provision of electronic stores makes comparative shopping possible, allowing customers to browse, compare, and order goods selectively. From the entrepreneur’s viewpoint, the variety of available applications generates new business opportunities for providing new services by integrating, enhancing, or customizing existing e-commerce applications:

1. Comparative shopping services are obtained by integrating several stores, providing the user with a uniform interface for posing requests, and having the application interact with the different stores to find the best bargains (see e.g. [50, 42]).

2. Integration of complementary services may also be useful. For instance, existing airline, train, car rental, and hotel reservation systems can be combined within one application offering an integral traveling service.

3. Existing applications may also be customized for special needs. For example, an adult bookstore can be obtained from a regular bookstore by supplying a wrapper that provides the appropriate user interface, restricts the search to the above category, and possibly gives some additional facility for hiding the customer’s identity.

Unfortunately, the diversity of applications in each specific domain and the disparity of interfaces render the integration and manipulation of applications a rather difficult task: Application data (e.g. store catalogs, form attributes) may have different formats, making the collection and comparison of data complex. Also, the application flow, the roles of the various actors participating in the application, and the API may vary widely among applications, complicating the coordination of activities between applications.
The situation in 2000, was that there was no single standard. Indeed in previous years there had been an intensive effort to develop standards for e-commerce applications in specific domains [22, 41, 59, 47]. This however provided only a partial solution: (i) Customization of existing applications to the new standards faced essentially the same challenges as in item 3 above. (ii) Although more uniform, the new standards obviously still left some design freedom to the application developers. (iii) One may still want to utilize existing useful applications even if they don’t conform the new standards.

A satisfactory solution to the integration and customization of e-commerce applications has to deal with the application flow and interaction between the various parties participating in a business transaction, as well as with the data involved in the process. In recent years there has been a significant amount of research on data integration and customization (see for instance, for a very small sample, [27, 37, 28, 90, 71, 105, 49]). However, the focus of these works has been on querying the data available in Web applications. The additional services offered by the applications, the roles of the various actors participating in the business transaction and the interaction between them, and the application flows, were mostly ignored (except perhaps for some description of the screen sequence needed to be traversed to obtain the searched for data). In contrast, the Application Manifold system (AM for short) presented in this thesis offers an integral solution to the integration/customization problem, covering in one framework, both data and operational aspects the of e-commerce applications. The system provides:

1. A novel declarative specification language for specifying the integration and customization task; then, acting as an application generator, the system generates a complete integrated/customized e-commerce application, with the declarativity of the specification allowing for the optimization and verification of the generated application,

2. An infrastructure based on XML [129] and UML (the Unified Modeling Language) [110], for the modeling of the applications data and flow, resp.,

3. A modular approach where new applications can be added and integrated at
The specification of an AM application then has two parts: The first models the target integrated/customized application with which the user will interact. (We will refer to this application as *global*.) The second describes the actual underlying Web applications and their relationship to the global one. (We will call these applications *local*).

A key observation is that local applications can be modeled as special *views* of the global application. This is in a sense an adaptation of the *Information Manifold* (IM) paradigm [90] used for data integration, to the context of e-commerce applications integration (hence the name *Application Manifold*). In IM, local data is modeled as a traditional view of the global data. Then, queries on the global data are processed by rewriting into queries over the local data. A significant difference here is that while IM focuses only on *data* and *queries*, AM covers the full profile of e-commerce applications: at execution time, *all* the operations/interaction of the various actors participating in the global application are rewritten into relevant processing in terms of the local applications, (possibly with some additional global processing). This of course requires the AM view specification language to go beyond the traditional data-oriented view languages and handle all aspects of the application and not just the data. Consequently, AM introduces a novel rewriting paradigm, enhancing the traditional query rewriting with a *refinement* mechanism dealing with the operational aspects of the application. A major advantage of the approach is that the addition of a new local source to an integrated application is rather convenient, typically requiring only the specification of the source view in terms of the global application (and supplying a corresponding wrapper for the source.)

**Limitations** The scope of work in this chapter is limited to web-enabled e-commerce applications. We do not support the integration/customization of proprietary/legacy applications. Wrapping of such applications as web services is complementary to our work. Choosing to describe local application as views over a global application (see also discussion in Section 5.3) has some tradeoffs. The main advantage is that by using the global application as a base model and describing local applications in the way they relate to this model, we actually define the semantic and glossaries of
the integrated application in one coherent framework. No further mapping between
data or methods is needed. However, a resulting limitation is that one can only use
data and method of the local applications that are specified in the global model. For
example if a bookstore offers a special feature as publishing the TOC, which is not
included in the global model, it will not be available for users of the global integrated
application. Similarly, the granularity of the global application building blocks can-
not be more refined than that of the local ones. For example if the global application
offers a purchasing process consisting of two separate steps, (1) submission of delivery
information, and (2) payment, it will not be possible to use a local application where
the two actions can only be performed simultaneously in a single screen/command.
Thus, to utilize as many local applications as possible, the structure and granularity
of the global application should be specified carefully using the available knowledge
on the relevant local applications.

This chapter is organized as follows. Section 4.2 introduces the underlying data
model, query language, and application flow model for the specification of global and
local applications, and describes the system architecture. In Section 4.3 we present our
running example and use it to illustrate various components of an AM application
specification. The semantics of such a specification is then explained in Section 4.4.
The system implementation is discussed in Section 4.5. In Section 4.6, we revisit the
results with current perspective, and show how the implementation can be modified
to use the BPEL standard. Finally, we summarize this chapter in Section 4.7.

4.2 Preliminaries

Handling e-commerce applications from different sources requires a common frame-
work in which the various applications and their data can be presented and modeled.
As demonstrated in [1], a typical e-commerce application can be abstractly modeled
by specifying the kinds of actors participating in the application and for each actor
describing: (i) the data available for the actor and its access rights (ii) the possible
operations (methods) on the given data, (iii) the various activities that the actor may
be engaged in, with the subset of data/operations relevant for each activity, and (iv) the application flow. To illustrate these components consider an example e-commerce application; an electronic bookstore. Typically such a store involves several types of actors, e.g., customers and vendors. It also involves a significant amount of data, e.g., the book catalog (typically searched by customers) or the promotion information (typically viewed by customers and updated by vendors). Observe that each of the actors may utilize different parts of the data (e.g., a customer can only see his/her own orders and the promotions relevant to his/her category, while vendors may view all the orders and promotions), each may have different access rights for the data (e.g., promotions can be updated only by certain vendors), and may perform different operations. Each actor typically performs several activities, e.g., a customer may be searching the catalog, ordering books, changing a passed order. Observe that in each of these activities, the actor may be facing a different Web page that possibly includes only part of the data and operations available to that given actor. Observe also that actions performed by an actor in a particular activity may initiate other activities/actions. For instance, when a customer orders a product, we may want to update the stock; when promotion is updated, we may want to refresh the customer’s screen with the new data.

The form of the general specification of an application (both global and local) is illustrated in Figure 4.1.

<table>
<thead>
<tr>
<th>Application application_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor actor-kind_1 in application application_name</td>
</tr>
<tr>
<td>data specification</td>
</tr>
<tr>
<td>methods specification</td>
</tr>
<tr>
<td>activities specification</td>
</tr>
<tr>
<td>application flow specification</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Actor actor-kind_n in application application_name</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Figure 4.1: Application specification

The concrete syntax used for each part of the specification of global and local
applications is given in the following sections. Readers familiar with [1] will notice that to some extent our AM syntax resembles that of [1] — after all both systems are used for constructing e-commerce applications. It is important to note however that the goal, hence semantics here is different: In [1] specifications are used for constructing new e-commerce applications from scratch, with data and methods coming from a given database system (e.g. a new electronic bookstore with a catalog and customer information stored in a specific database). Hence the data part for example, describes which part of the given database is viewed by each actor, and the methods part provides the code for the methods available for the actor. In contrast, AM specifications describe the integration and customization of existing Web applications (e.g. the integration of several existing electronic bookstores into one giant bookstore). Hence the data/methods/activities/flow parts of the specification describe how the data/methods/activities/flow of the integrated global application are realized in terms of the corresponding local ones. To prevent confusion it should also be noted that while both [1] and AM use the term view, the context and usage is completely different: While the first uses traditional database views for describing the portion of the database that is available for the actors participating in the application, AM uses application views for modeling the local applications as views of the global integrated application.

Before describing the syntax and semantics of the language, we give some preliminary background and examples of (1) the XML structure used for modeling the data and (2) UML and the State Charts formalism that will be used throughout this chapter for the modeling of application flows. Then we present the architecture of the Application Manifold (AM) system which relies on the above.

### 4.2.1 Data Model and Query Language

Since our goal is to support Internet applications such as e-commerce, we model the applications data using XML [129], and use an XML query language for describing the data portion of the applications view. We now demonstrate the XML documents and DTDs that will serve as a running example in the rest of the chapter. Full
definitions of the XML query language can be found in [51]. Figure 4.2 shows a fragment of an XML document describing a book catalog summarizing information from several Web bookstores. Each category item consists of a sequence of sub-items such as Category_Name, Book, etc.

Figure 4.3 shows a possible DTD for the document in Figure 4.2. We will use in the sequel the attribute name ACCESS to specify the allowed access rights for the given data items. (Here Reviews can be read or added but not deleted/modified.) For brevity we will assume that the default access right for an item is read only, and will explicitly specify the attribute only when different from the default. In general, typing is not a mandatory feature in XML, however, since most optimization techniques rely on typing, it is realistic to assume that large XML applications will come with appropriate DTDs. In the sequel, we will denote element type definitions using the Elem suffix. For instance, CatalogElem will denote the type definition associated with the catalog element of the DTD given in Figure 4.3.

When this work was written there was no standard query language for XML. We rely here on XML-QL [51], which is very similar to XQuery. The mapping from the queries presented in this chapter to XQuery is straightforward. For example, the query in Figure 4.4 searches the global books catalog for books sold by
4.2. PRELIMINARIES

Figure 4.3: The Catalog DTD

```xml
<!ELEMENT Catalog (Category *)>
<!ELEMENT Category (Category_Name,Book *)>
<!ELEMENT Book (ISBN,Title,Authors,...,Stores)>
<!ELEMENT Authors (Author *)>
<!ELEMENT Stores (Store *)>
<!ELEMENT Store (Storename,Price,...,Reviews)>
<!ELEMENT Reviews (Review *)>
<!ATTLIST Reviews ACCESS (read,append) #IMPLIED>
<!ELEMENT ISBN #PCDATA>
...
<!ELEMENT Review #PCDATA>
```

Figure 4.4: XML-QL Query

"BookStore 1" and constructs the store's catalog: The WHERE clause describes the pattern to be searched for in the data. The relevant items are bound to the variables $X,$Y,$Z,$P,$Q. Then, for each such variable assignment, a book element with structure as is the CONSTRUCT clause is constructed. The nested query there basically “flattens” the Reviews item, obtaining book items with structure

```xml
<!ELEMENT BookStore1_Book (ISBN,Title,Authors,Price,Category_Name,Review *)>
```

For this work we used only a fragment of the language, focusing on simple paths only. This simplifies the task of data integration (to be explained later) and seems sufficient for the applications that we considered. Another feature supported by XML-QL are Skolem functions in the CONSTRUCT clause, typically used for the grouping of several elements into one element. Since grouping can also be archived using nested
queries\[51\], we will assume below that our views use only nested queries.

### 4.2.2 Application Flow

We use *State Charts* [77] to describe the application flow and to capture the fact that actions performed by an actor in a particular activity may initiate other activities/actions. ¹ Besides providing a rich and flexible visual formalism where user activities can be modeled, State Charts support a refinement mechanism (as described in Chapter 2) that provides convenient means for the modeling of the integration of applications. State Charts are also part of UML (Unified Modeling Language) [110], the emerging standard for modeling of applications, hence, together with XML, provide a solid basis for our work.

As mentioned in previous chapters, a state chart is a higraph (for detailed description see [77]) consisting of nodes representing states (denoted here by rounded rectangles), possibly nested and linked by transitions. Figure 4.5 illustrates the notation we use here for state charts. The state names, when significant, are written in the small rectangle attached to the top of the state (e.g. *Search.Activity*, *ShoppingCart.Activity*, *Search.Method*). The arrow emanating from the black dot signifies an initial state. A transition is denoted by an arrow and labeled by the name of the event causing the transition (e.g. *goto/Search.Activity*, *search*, *browse*). They can also have an associated guarding condition - a boolean expression that evaluates to True or False. The transition occurs only if the guard is True. Guards are written within square brackets following the event name (e.g. *search/[valid(Search.Params)]*). A state may have associated actions that are executed on entry to the state or upon exit from the state, e.g. when entering the *Search.Method* state, the method *search()* is executed; then the state is exited, and the event *goto/ShoppingCart.Activity* is signaled (causing transition to the *ShoppingCart.Activity* state). Parallel execution of activities can be described using orthogonal states, graphically separated by dashed lines (see, e.g. Figure 4.22, to be discussed in detail later on). State Charts also support a

¹In [1] active rules (triggers) were used instead. We have chosen here State Charts since they generalize the active rules in several ways and are more suitable for the modeling of applications integration.
4.2. PRELIMINARIES

refinement mechanism that allows simple non-composed states (e.g. Search_Method) to be associated with a detailed state chart describing the actual implementation of the component, hence providing a modular description applications.

The UML State Charts semantics [110] that we use here assumes that a state machine described by a state chart reacts to events applied to it by some external objects. Event processing by the state machine is partitioned into steps, each of which is caused by an event directed to the state machine. The fundamental semantics assumes that events are processed in sequence, where each event stimulates a run-to-completion (RTC) step. The next external event is dispatched to the state machine after the previous step is completed. Once an event instance is dispatched, it may result in one or multiple transitions being enabled for firing. By default, if no transition is enabled, the event is discarded without any effect.

4.2.3 The Application Manifold Architecture

The AM system is based on a client/server architecture. An AM application consists of several independent clients communicating with the AM server and possibly between them (via the server) using an XML-based API. Figure 4.6 shows the various components of one application (obviously, several such applications may run simultaneously on the server).
As mentioned above, an AM application consists of (1) a (virtual) global application with which the user interacts, and (2) the (actual) local applications implementing it. The task of the AM is to accept a request from clients of the first kind and process them by communicating with applications of the second kind. The communication with the underlying Web applications is done via wrappers that translate the standard AM API requests to application-specific http calls (and in the opposite direction mappings of the obtained data to the corresponding AM XML-based representation).

The AM consists of five main components, sketched below.

The *request parser* receives user requests (messages in XML format), parses them and passes the constructed request objects to the *request rewriter*. This consults the specification of the given application, and rewrites the request into a corresponding state chart, expressing the request in terms of the local applications. The *validation module* verifies the correctness of the construction (e.g. the capabilities of the various sources and the adequacy of the passed parameters). The state chart is then passed to the *application manifold engine* for execution. The engine is in charge of communicating with the various sources, passing and receiving data and requests and maintaining information about the current state of each application. The *results integrator* provides data integration services for the data accumulated in the processing.
The engine is also in charge of communicating with the user, passing request results and relevant notifications. The request rewriter and the application manifold engine use an XML repository (omitted from the figure): The AM application specifications are stored in the repository and the request rewriter consults it for selecting the components relevant to a given request. Similarly, the repository is used by the AM engine to store and manipulate the data and state information accumulated in the processing.

4.3 Example

We will illustrate things throughout this chapter using a simple running example - a global book store which integrates services offered by several bookstores on the Web. Note that although it is similar to some of the applications available on the Web for comparative book shopping (e.g. [50, 137]), our emphasis here is different: our goal is not to present yet another comparative shopping application but rather to demonstrate the core technology behind the AM system, which enables a declarative specification and generation of arbitrary integrated/customized e-commerce applications (with comparative shopping being just one possible instance). We present below the application specification. The semantics of such a specification is then explained in the following section. Our AM specification will consist of two parts, the first describing the global bookstore application with which the user interacts and the second describing the (relevant parts of the) actual underlying local bookstore applications and their relationship to the global one. We start with the specification of the global application.

4.3.1 The Global Bookstore

We define for each actor type participating in the global application (i) the data available for the actor, (ii) the possible operations on the data, (iii) the activities that the actor can be engaged in, and (iv) the application flow. We detail below the global Customer specification. Other actors (e.g. suppliers, publishers) can be
Data and Methods Assume that the data available to a customer includes: 1. a catalog with a structure as in Figure 4.3 which can be browsed and searched using the methods browse and search resp., 2. the search parameters (used as input to the search method), 3. the search result, 4. a shopping cart where items can be added or deleted, and 5. a list of the customer pending orders with the methods order to place a new order for the items accumulated in the shopping cart and status to check the delivery status of a pending order. The data and operations specification for the customer actor is partially given in Figure 4.7. All the data variables except the second one are common, in the sense that their values correspond to (possibly integrated) values originating in the underlying local applications (e.g. the catalog presented to the user will in fact be the integration of catalog data from the local bookstores. Similarly the global shopping cart is the union of the customer’s carts in the various stores). In contrast, search_params is private in the sense that its value is supplied by the user (and may be passed as input to the underlying applications). Similarly, the browse, search, and order methods here are defined as common: their actual code is not specified in the ActiveView; instead they will be implemented by communicating with the underlying applications, issuing (possibly a sequence of) local methods. In addition to the methods listed explicitly in the specification, we also have read/write/append/... methods for the various data elements, according to the access rights specified in the element DTD.
### Activities and Flow

Assume that a customer can be engaged in four activities: (1) login into the system, (2) searching the catalog, (3) adding/removal of books from the shopping cart, and (4) placing an order. From an end-user viewpoint, each activity corresponds to a Web page with some data and buttons. For instance, the `Search Activity` page will show the catalog and the search criteria (to be filled in by the user), and some buttons allowing the user to call the search and browse methods, change activity or quit the application. The `Shopping Cart Activity` page will show the search result and the shopping cart, and some buttons allowing the customer to add/remove elements from the cart or switch to the ordering activity. This is specified as in Figure 4.8.

Finally, the application flow is defined using a state chart: The default state chart associated with an actor has one state per activity, entered when `goto_ActivityName` signal is signaled. Upon entry the values of the activity variables are read. Each activity has sub-states representing the activity methods. They are entered when the relevant method button is pressed, and the relevant method is executed upon entry. One can customize this default flow by adding components or refining the flow. For example, Figure 4.5 shows part of the default customer state chart (for brevity some of the methods and activities are omitted), customized by adding a guarding condition to the search method (verifying before execution that the search parameters are valid). Also, once the search is executed, the customer is automatically switched to the shopping cart activity to be able to add some of the selected items.
4.3.2 Local Bookstores

Local applications are defined as views over the global application. Then, at execution time, the user requests and flow of the global application will be rewritten to actions in terms of the local applications, (possibly with some additional global processing).

Assume we are given several local applications which we will call in the sequel BookStore1, ..., BookStoreN. We detail below BookStore1 (a somewhat simplified version of the Barnes&Nobel Web site). The other applications (corresponding e.g. to Amazon.com and alike) are treated similarly. Upon entry to Bookstore1, the user can search the catalog by one of three criteria: title, author, keyword. To activate other types of searches, the user clicks on an ‘advanced search’ button getting a Web page where he chooses (again, by clicking on the relevant buttons) between two possible Web pages: the first allowing a search by ISBN, and the second allowing a search by one or more of the criteria, author title and keyword, optionally restricted by price, format, age, and subject. The various searches return a search result including only a short description of the retrieved books. To get more information on a specific book, users need to click on a ‘read more’ button next to the book, switching to a page containing the full book data, where they can click an ”add to shopping cart” button, adding the book to their cart.

We next define this local application as a view over the global application. It should be noted that only the relevant parts of the local application need to be specified: the application may feature other data/activities that are not covered by the global application; these need not be specified.

Data The local customer data here includes, among others, the catalog, the three types of search criteria, the search result, and the shopping cart. The structure and access rights are defined by a local DTD (omitted here) and may differ from that of the corresponding variables in the global application. The variables value is defined as a view (query) over the global data, basically projecting out the data originating from the given application. This is illustrated in Figure 4.9 below, where Query1 is the query of Figure 4.4, retrieving the BookStore1 portion of the global catalog, and Query2 is the query in Figure 4.10, retrieving the ‘quick search’ parameters
4.3. EXAMPLE

let bookstore1_catalog: BookStore1CatalogElem be Query1
let quick_search_params: QSearchParamsElem be Query2
let ISBN_search_params: ISBNSearchParamsElem be ...
let full_search_params: FSearchParamsElem be ...

Figure 4.9: Variable definitions in the local application

<QSearchParams>
  WHERE <SearchParams> <$> $Q</> 
  IN "GlobalBookStore/SearchParams.xml"
  $P in {Title,Author,Keyword}
  CONSTRUCT <Criteria>
    <$> $Q </>
  </>
</>

Figure 4.10: Query2

(title,author,keyword), if they exist, from the global search criteria. In general each local variable is defined as a query over one variable of the global application. Recall that we distinguished there between common and private variables (e.g. catalog vs. search_params). Following the same terminology we will refer to the local variables defined by queries over common (private) variables as common (private).

In addition to the above data variables, which correspond to variables of the global application, the local application may use some additional local variables: Web applications often maintain information about the user’s state/context in cookies and variables that are passed to the user’s browser and sent back to the application server with each user request. To continue with our example, assume e.g. that BookStore1 also contains (as in the original Barnes&Noble application) the local variables userid: UidElem and log:LogElem. (Here we do not have associated queries).

Methods, Activities and Flow Once the local data and its relationship to the global data is specified, we continue with the methods, activities, and flow. The specification of a local application (just like that of the global one) lists the application’s methods, activities and flow, in a similar syntax. (As in the case of data, only the
methods/activities/flow relevant to the global application need to be specified.) The relationship between the global and the local ones is specified via the State Charts refinement mechanism: First, with each global activity $A$ we associate some local activity $l(A)$. Now, recall that each of $A$’s methods was represented in the state chart of the global applications by a (simple) state $A_m$. These states are now refined to include the relevant subset of the local application’s state chart, augmented with some additional control describing the specific flow required for capturing the global method, and having $l(A)$ as the initial and final states.

**Example 1:** To illustrate this consider for example the search method of the global Search_Activity in Figure 4.5. Figure 4.11 below shows a possible refinement of state Search_Method, with $l($Search_Activity$) = $Quick_Search_Activity$. The bold letters and solid lines of the refinement describe the original relevant portion of BookStore1’s state chart, while the dotted lines and italic letters describe the additional specific control implementing the local version of the global method. Recall from the beginning of the subsection that BookStore1 has three possible search activities (quick, ISBN, and detailed), with the user starting at the quick search screen and, if desired, moving to the other two types. Also recall that the three corresponding search parameters were defined above as views (queries) over the global search parameter. The added control here basically validates the value of the three possible local search parameters (i.e. the result of the corresponding view queries) w.r.t their DTDs, and executes the first applicable local search. Now, since the search in BookStore1 returns only a short description of the retrieved books, and since further details can be obtained only for one book at a time, the added control also includes a new state (More_Details_Activity), besides the original BookStore1 states, realizing a ‘read more’ iteration over the retrieved books.

Upon completion we go back to the initial state, ready to execute additional activity methods.

**Example 2:** To see a less ‘query-oriented’ example, consider the ShoppingCart_Activity and a method add_to_cart that adds a list of books to the cart. Since in BookStore1 only one book at a time can be added (and this can be done only once the book
Figure 4.11: Refinement for Search Method
was searched for and its full details requested and displayed), the refinement will include an iteration (similar to the one above) over the sub-list relevant to BookStore1 (defined as explained above as a view over the global list), with each loop realizing a sequence of BookStore1 activities: retrieving the given book (e.g. via an ISBN search), obtaining its full details, and then adding it to the actual (local) BookStore1 cart (see Figure 4.12).

**Example 3:** Note that while the examples above relate local and global actors playing essentially the same role (customer in both cases), this is not obligatory: one can use the same mechanism to relate global and local data/activities of arbitrary types of actors. For example, in our global application a customer may add reviews to the catalog. If in BookStore1 only certain vendors are allowed to modify the reviews, the customer’s global ‘add reviews’ method can be associated with the relevant local vendor’s activity, and refined by a state chart describing the necessary flow to be performed on behalf of the vendor. Furthermore, one can realize a global method by a combination of activities involving several types of local actors.

**Example 4:** Our framework can also be used to enhance exiting applications with new services. For example, assume that to assist users in deciding where to buy their books we want to (1) give the users a ‘cheapest bargains’ summary of the search result, and (2) record for each user her order history, allowing her to annotate the list with comments regarding the (dis)satisfaction of the given services. Now our global application has real coded methods in addition to those defined by refinement (e.g. a method cheapest_books, calling search and then querying the obtained search_result to find the cheapest stores), and real data stored in the system’s XML repository, in addition to the virtual data coming from the underlying applications. Variables in the global application can now be defined as views over this repository data, e.g.

```perl
let customer_order_history: AnotatedOrderElem* =
  WHERE <AnotatedOrd><Usid> $U </></> ELEMENT AS $X
  IN XML repository/OrdHist.xml
  $U =self.id
CONSTRUCT <AnotatedOrd> $X </>
```

User requests for real data/methods are then served by the XML repository, while the virtual ones are delegated by the AM engine to the underlying Web applications.
4.3. EXAMPLE

Figure 4.12: Refinement for Add_to_Cart_Method
Remark  One may wonder how difficult is it to model local application and define the above refinements. It is important to note that specifications do not require any knowledge of underlying implementation of the local applications, but only a rather simple analysis of their Web interface: Web pages or frames are naturally modeled as activities, page data and form fields as data variables, and buttons as methods. The flow in the refinement is then essentially a description of a typical usage of the Web interface.

In the next two sections we explain the semantics of the specification. Then we give a detailed example of the modeling process and the run time operation in Section 4.5.

4.4 How Things Work

Now that we have illustrated how AM applications are specified, we need to explain the semantics of such a specification. As shown above, the specification of the local applications consists essentially of two parts, the first modeling the local data variables as views (queries) over the global variables, and the second modeling the local application flow as a refinement of the global one. We will start by considering the more standard part - the data variables. Next we will consider the flow and explain how the two parts work together.

4.4.1 Data

Applications contain two types of variables, private and common. The value of private variables (e.g. search_params) in the global application is supplied by the user. The value of the corresponding variables in the local applications (e.g. quick_search_params, ISBN_search_params, full_search_params) is computed by simply evaluating the queries associated with the variables.

For common variables (e.g. catalog), the computation flows in the other direction: Their values originate from the corresponding variables in the local applications (e.g. bookstore1_catalog in Bookstore1). Then, following the Information Manifold
approach [90], the value of the corresponding variable in the global application is defined to be the \textit{maximal view rewriting} for the variable, given the views in the local applications (which basically means that the global variable contains as much relevant information as can be obtained from the corresponding local variables.)

Before explaining this further, let us first recall the standard definition of \textit{maximal view rewriting} from the relational data model [56]. Given a relation $R$, some views $V_1, \ldots, V_n$ over $R$, and a query $Q$ over $R$, the maximal view rewriting for $Q$ is a query $Q'$ that (1) uses only the views as EDB relations, (2) is contained in $Q$, and (3) contains any other query $Q''$ that satisfies (1) and (2).

Intuitively, $Q'$ computes as much as possible of the answer to $Q$ based on the information given in the views. To see the connection to our XML context observe that (1) XML documents are often modeled as edge-labeled graphs (with nodes representing the document elements, edges representing the “component of” relationship among elements, and with the edges labeled by the type of the pointed item, or with a data value, for the atomic leaf elements) [4, 51]. (2) Such graphs can be described by an \textit{edge} relation $R(from, label, to)$, with queries being modeled as mappings of such input edge relations to output ones.\footnote{The order among sibling elements can be captured by an additional order relation. To simplify the presentation we ignore this here.} Now, let $u$ be some common variable in the global application (with corresponding edge relation $R_u$) and let $u_1, \ldots, u_n$ be the corresponding source variables in the local applications, described as views over $u$ (with edge relations $R_{u_1}, \ldots, R_{u_n}$). The value of $u$ can be defined to be the maximal view rewriting of $R_u$ using $R_{u_1}, \ldots, R_{u_n}$.

Note however that we are not interested here in constructing arbitrary maximal XML documents for our global variables: First, we are primarily interested in \textit{valid} documents - Recalling that each variable has an associated DTD, we want the obtained documents to be valid w.r.t this DTD. Second, we may be interested in documents that obey some additional structural restrictions - for example, in our global book catalog, we want to have a unique category entry for each category name. (i.e. no two \texttt{Category} elements can have the same \texttt{Category\_name} value). Consequently, what we really want is a rewriting that is maximal w.r.t documents.
satisfying the constraints.

Again, to understand this, let’s go back to the relational model, where maximal view rewriting has been studied for databases obeying constraints in the form of functional dependencies [57]. To capture this, the definition of maximal view rewriting was adjusted so that query containment is tested only w.r.t databases obeying the given dependencies. Interestingly, many of the constraints we are interested in here can be viewed as functional dependencies between document elements. For example, a DTD requirement, say, that a ⟨Book⟩ element contains one ⟨ISBN⟩ sub-element, amounts to the fact that the identity of the ISBN element in the document graph functionally depends on the id of its parent book item. Similarly, in our catalog, the identity of the ⟨Category⟩ node functionally depends on the value of its ⟨Category_Name⟩ child.

Modeling our document constraints as functional dependencies allows us to harness the techniques from the relational world for computing our global variables. We will focus here on the above two types of functional dependencies, the first modeling a DTD requirement for a given element to contain only a single occurrence of a sub-component of a certain type (hence the identity of the subcomponents functionally depends on the id of its parent), and the second, modeling cases where the id of an element functionally depends on the id of some of its subcomponents. Figure 4.13 describes the algorithm for computing the value of a global variable. We explain it below and illustrate it with an example.³

**Step 1: Inverse queries** It was observed in [56] that, in the case of relational conjunctive views and queries, the maximal view rewriting can be obtained by generating for each view some inverse rules computing the portion of the global relation that is captured in the view. The key observation is that since we use here only a fragment of XML-QL including in particular, only simple paths, our views are essentially conjunctive queries over the edge relation, and hence similar inversion principles as in [56] apply. Adapting this to our context we generate for each local variable \( u_i \) defined

³Clearly not all constraints imposed by a DTD can be modeled by such functional dependencies. The problem of maximal view rewriting that fully captures document validity is an open problem, see section 5.3.
4.4. HOW THINGS WORK

Algorithm: Global variable computation

**INPUT**  
AM specification, set of functional dependencies, local variables $u_1, \ldots, u_n$

**OUTPUT**  
global variable $u$

Step 1: For each variable $u_i$ defined by a query $Q_i$, construct an inverse query $Q'_i$.

Step 2: Annotate items in $Q'_i$ with Skolem functions to capture the functional dependencies.

Step 3: For $i = 1 \ldots n$, compute the annotated $Q'_i$ on $u_i$.

Step 4: Merge the results to obtain a canonical value for $u$.

Figure 4.13: Global variable computation

by a query $Q_i$ an *inverse query* $Q'_i$ by basically switching the roles of the WHERE and CONSTRUCT clause of the query. There are two delicate points here.

- First, the WHERE clause may contain some variable names that do not appear in the CONSTRUCT. (This is analogous to the case where the body of a conjunctive query contains some variables no appearing in the head). As in the case of conjunctive queries, when the query is inverted and the roles of the WHERE and the CONSTRUCT clauses are switched, those variables are simply replaced in the new CONSTRUCT clause by a corresponding Skolem function of the remaining variables [56].

- Second, we need to handle nested queries. Recall from Section 4.2.1 that a nested query is essentially composed of two parts. A variable in the WHERE clause that defines the scope of the query (e.g. the variable $P$ in the query in Figure 4.4), and nested query over this variable in the CONSTRUCT clause. When inverting the query we also invert the role of these two parts: The nested query in the old CONSTRUCT clause (which now becomes a WHERE clause) is replaced by a new variable name whose scope is the item where the nested query appeared, and the (inverted) nested query is moved to the new CONSTRUCT (previously the WHERE), replacing the old scope variable.

To continue with our example, the inverse of Query1 from Figure 4.4 is given in Figure 4.14 below. Ignore for now the ID attribute of the elements. (this will be explained below). The WHERE and the CONSTRUCT clauses of Query1 are switched, and
the nested query (with its WHERE and the CONSTRUCT clauses also switched) is moved to the (new) CONSTRUCT clause, being switched with its source variable \$P.

**Step 2: Skolem functions**  To capture the restrictions imposed by the given functional dependencies, we associate with each restricted element \(\langle E\rangle\) in the CONSTRUCT clause a Skolem function \(S_E\) determining the element ID as a function of the elements on which it depends. Again, this is an adaptation of the relational treatment for functional dependencies of [57], equating elements that functionally depend on the same values: Skolem functions are used in XML-QL to control how the result is produced and grouped. In Figure 4.14, whenever a \(\langle Category\rangle\) is produced, its associated ID is \(S_{\text{Category}}(\$$C$$)\). \(S_{\text{Category}}(\$$C$$)\) is a Skolem function, and its purpose is to generate a new identifier for every distinct values of \$$C\$. If, at a later time, the query binds \$$C\$ to the same value again (e.g., by finding another book of the same category), then the query will not create another \(\langle Category\rangle\) element, but instead will append information to the existing \(\langle Category\rangle\). Thus, all the \(\langle Book\rangle\)'s of that category will be grouped under the same \(\langle Category\rangle\) element. The \(\langle Category\_Name\rangle\) element has a single occurrence since its id depends on the category id. Note that in this example the only non-constraint elements are the \(\langle Review\rangle\)'s. They thus have no associated Skolem function and a new \(\langle Review\rangle\) is produced for each distinct assignment for the query variables.

**Steps 3 and 4: Variable computation**  Now, given the inverse queries for all the variables in the local applications, we construct a single canonical value for the global variable \(u\). This is achieved by evaluating the inverse queries for all the \(u_i\)'s and “fusing” all the elements with the same id into a single element. For instance, in the above example, for each book, all the catalog data coming from the various local applications will be grouped under the same \(\langle Book\rangle\) element (since the id of the book element is defined to be a Skolem function of the book’s ISBN, which is the same in all sources).
4.4. HOW THINGS WORK

Example of Data Integration To illustrate the actual data integration process, assume we have two book stores, BookStore1 and BookStore2, each with its own books catalog, and consider their integration into a global catalog. As before, assume that the relationship between the global catalog variable and that of BookStore1 is defined using Query1 (appearing in Fig. 4.4), and let Figure 4.15 describe part of the actual catalog content of BookStore1.

Next, consider the catalog variable of BookStore2. Naturally it may have a different structure than that of the first book store, so assume for example that it does not contain book reviews. The relationship between the global catalog variable and that of BookStore2 is defined using the query in Figure 4.16. As we did for BookStore1, we define a corresponding inverse query for the variable (see Figure 4.17). Notice that as this source contains no Review items, the query (and thus its inverse too) is not nested. Finally, Figure 4.18 describes part of the actual content of BookStore2’s catalog.

To conclude the example we need to see how the two catalog are integrated, namely execute steps 3 and 4 of the algorithm. We first apply the two inverse queries on the corresponding local catalog variables. The results are given in Figures 4.19.
and 4.20 respectively. Notice that the Skolem-based ids assigned to the items are important even before the results are integrated: for instance, the output of each of the two queries contain only one Catalog element rather than the two that would be generated if no Skolem functions where used. Similarly, the result of the second inverse query (Figure 4.20) contains a single Category element with the name Computers rather than the two that would be created in the absence of Skolem functions.

Finally, we run the forth step of the algorithm and merge the results of the queries into an integrated global catalog by fusing elements with identical id’s. The result is described in Figure 4.21, where, for clarity, the identifiers are omitted from the integrated catalog, (as they are not really part of the data and are only used for grouping and fusion.) Intuitively, the integrated catalog is the maximal one could obtain given the sources - it contains all the available books information without redundancies.

**Remark on consistency** The above algorithm assume the consistency of the data sources w.r.t the given set of functional dependencies. In practice, although each source may individually obey the constraints, (e.g. books ISBN determine their title), they may not be preserved globally due to errors or different writing standards, (e.g.
4.4. HOW THINGS WORK

Figure 4.16: Query 2: BookStore2’s Catalog as a query over the Global Catalog

a book with the same ISBN may have distinct titles in different stores due to spelling mistakes or shorthand notations.) Fusion (in step 4) of such inconsistent data may produce multiple values for an item, rather than the single one required by DTD. When this is the case, we present the alternative values to the user and let her/him chose. Automatic resolution of inconsistencies[113], based e.g. on linguistics analysis and source reliability, can easily be incorporated.

Remark on optimization  The current default in the system is that the global variables are fully computed when a read for the variable is requested (see below). For certain applications this may be redundant: Consider for example the global book catalog. When browsing the catalog, the user will typically visit only a few entries and then will stop either because the needed data is found or because she decides to call the search method for issuing a particular search. In this case it is better to view the above variable value definition as virtual and employ lazy evaluation [97], computing the relevant elements only as the user navigates into the data. To overrule the default, one can use two specific kinds of read modes, namely, deferred read or immediate read. The first instructs the system to compute elements only on demand, while the latter indicates that elements should be computed immediately when encountered.
WHERE <BookStore2_Catalog>
  <BookStore2_Book>
    <ISBN> $X </> <Title> $Y </> <Authors> $Z </> <Price> $Q </> <Category_Name> $C </>
  </>
</>
IN "BookStore2/bookstore2_catalog.xml"

CONSTRUCT <Catalog ID=S_Catalog()>
  <Category ID= S_Category($C)>
    <Category_Name ID=S_Category_Name(S_Category($C))> $C </>
    <Book ID=S_Book($X)>
      <ISBN ID=S_ISBN(S_Book($X))> $X </>
      <Title ID=S_Title(S_Book($X))> $Y </>
      <Authors ID=S_Authors(S_Book($X))> $Z </>
      <Stores ID=S_Stores(S_Book($X))>
        <Store ID=S_Store(S_Stores(S_Book($X)),''BookStore2")>
          <Storename ID=S_Storename(S_Store(S_Stores(S_Book($X)),''BookStore2"))>
            ''BookStore 2"
          </>
          <Price ID=S_Price(S_Store(S_Stores(S_Book($X)),''BookStore2"))> $Q </>
        </>
      </>
    </>
  </>
</>

Figure 4.17: The inverse query for Query2

4.4.2 Activities and Application Flow

Recall from section 4.2.2 that the activities and flow of each actor in the global application are modeled by a state chart, with user requests interpreted as events in the corresponding state machine. Now, at execution time, the State Charts refinement mechanism is employed: on each user request (method call), rather than activating the relevant global state, its local refinements are executed. Observe however that typically each global method (state) may have several such refinements, one per local application. In the computation, the multiple refinements are viewed as orthogonal states and performed in parallel.

To continue with our example, consider the search method of the global bookstore (modeled by the simple state Search_Method in Figure 4.5). A method call results in running the compound state machine in Figure 4.22 which includes $n$ orthogonal states representing the refinements of Search_Method for the local applications BookStore1,...,BookStoreN. Note that in general some methods may not be supported by all the local applications. For example, BookStore1 may support a catalog search but not an arbitrary browsing of its catalog (hence its specification may contain no refinement for the Browse_Method of BookStore1). The constructed state chart contains only the applicable refinements.
To see the connection with the AM system architecture, recall from Section 4.2.3 that the Request Rewriter module of the AM is responsible for constructing the state chart portion to be executed upon user request. The state chart being built for each method call is precisely the refinement described above, with each orthogonal state corresponding to one local application. In principle, rather than running all local applications in parallel, the request rewriter could decide, for optimization reasons, to serialize the requests to certain web sites as a result of validation checks or optimization considerations. For example, a serialization policy may be adopted to eliminate redundant execution if the necessary information is already fetched from other sites. While the current implementation is fully parallel and does not support this type of optimization, with some modifications to our system, it can be incorporated in our platform.

Once constructed, the refined state chart is passed to the AM engine for execution. Recall that in each such refinement, part of the state chart describes actual activities of the local applications (the solid lines and bold text in Figure 4.11), while part describes the added control needed for implementing the global method (the dotted lines and italic text in Figure 4.11). When the AM engine runs the state chart, the bold-text parts are executed by communicating with the local applications, while the control in the italic-text parts is executed by the AM engine itself.
In between consecutive user requests, the AM system maintains the application context, for both the global and local applications. This includes the application state and the current value of all variables for all the actors in the application. The variables and states are augmented with expiration time and stored in the XML repository. When a user issues a new request within a reasonable time to the given application, the relevant state machines resume the states they last had at any depth within the compound states and all variables values are restored. Then the request state machine is run starting from this point.

4.4.3 Putting It All Together

To conclude, we explain how the flow part and the data part work together. Recall from the previous subsection that we distinguish between two types of variables,
private and common, the first originating in the global application, while the second originates from the local ones. When a private variable needs to be passed to a local application, its value is computed (as explained above), from the corresponding global variable, and vice versa, when a read request for a common variable is issued by the global application, its value is computed from the corresponding local variables. The data in the global application is obtained by check-in/check-out, e.g. explicit calls to the global read and write methods. So in general changes are not immediately propagated. (But the freshness level of data can be controlled by the programmer by specifying in the DTD the desired read/write frequencies).

Read: Observe that the data in local applications may be updated from outside the AM system, e.g. via its direct Web interface. So each global read request is refined to a fresh local read of the corresponding local variables. Then the system computes the value. There are two possible optimizations that can be added to the system: (i) Some Web applications (e.g. [1]) support a notification mechanism which allows notifying
Figure 4.21: An excerpt of the global catalog after the integration)
the AM engine on changes to the local data. Recording such notifications will allow
the AM engine to avoid redundant reads. (ii) Rather then fully recomputing each
time the global value, we can compute the delta between the new and old local values
(using e.g. a diff algorithms as in [135]) and only propagate the change. Recall from
section 4.4.1 that we construct an inverse query for each local variable and compute
the value of global variables by merging the results of those queries. This allows us to
view the global variable as a view of the local data (although the original specification
was revered - local variable were defined as view of global ones!). So the problem at
hand is that of propagating updates from the database (the local data) to the view
(the global variable), which can be performed in the style of [3].

Write: When a variable in the global application is modified and a write is re-
quested, we have to propagate the change from the global application to the local
ones. The local data is defined as a view over the global data. By default, a global
write is refined to a local write with the new value for the view. Again, this can be
optimized. If the system maintains the delta between the old and the new value for
the global variable, an incremental evaluation of the views can be performed in the
style of [3]. Besides computation efficiency, incremental evaluation has two additional
advantages here:

1. Since we are in a client server architecture, sending the delta instead of the
   entire value may result in large saving in communication.

2. The second issue is access rights. Consider for example the variable catalog in
   the Customer global application, and assume that some reviews are appended
   to a given book and a write is issued for the catalog. Assume that the access
rights of $BookStore1\_catalog$, as declared in the DTD, is read for all elements but the $(Reviews)$ which can also be updated. An attempt to propagate the write by replacing the whole $(BookStore1\_Catalog)$ element by a new one will be rejected by the local application due to access rights violation, whereas an incremental update replacing only the $(Reviews)$ will be accepted.

Updates that cannot be propagated due to violation of access rights in the local applications are rejected by the system. Another difficulty is updates that are not relevant to any of the local applications. To see an example, assume that our application allows certain users to update all the catalog data. Such a user could in principle add book elements sold by a $(Storename)$ other than any of the given local applications. But now when a read is issued for the global catalog, a re-computation of its value from the local applications will cause the update to disappear. This could be anticipated by rejecting non-propagative updates and is planned to be incorporated in the next version of the system.

Remark: We conclude this section with a remark on transactions. The transactional support that a global application can give depends on the capabilities of the underlying local applications. By default, a global application is not in a transaction mode: even if one assumes that method calls in local applications are atomic, the local refinement of a global method may consist of a sequence of such calls. Transactional behavior can be achieved only if the local applications support explicit methods to start a transaction and terminate it with an abort or commit, with the level of global atomicity depending on the level of the available local support [7].

### 4.5 Implementation

The AM system is implemented as a mediator system that simultaneously plays two roles: a server for the global applications, accepting and handling user/application requests, and a client of the local applications, imitating a browser when connecting
to remote services.\textsuperscript{4}

To keep the application simple and open and to enable portability we mostly use public domain tools. In particular, we run the \textit{Apache Web Server} [10] with \texttt{mod_perl} [10] on a \textit{Windows NT} machine (these tools were also tested on a Linux machine and are also applicable to different Unix machines). The implementation work consists of three parts: (1) implementation of the AM system. (2) writing the specifications of the global and local applications. (3) writing of wrappers. We will describe each of these below.

\subsection{The AM System}

As explained in Section 4.2.3, the system consists of five modules (see the center part of Figure 4.6). They are implemented in Perl, C++ and Java in windows environment, and activated by a CGI script upon a user request. The communication with the users and the local applications is managed by the Apache Web server and proxy server resp. Recall that communication with the local applications may require the activation of an appropriate wrapper. For that we have replaced one of the modules of the proxy server with a \texttt{mod_perl} script of our own, in charge of wrappers activation.

\textbf{Request parser:} Implemented in perl. Parses the AM XML-based API (global) request and, using the stored applications specification, attaches the list of relevant local sources. The resulting object is passed to the validation module.

\textbf{Validation module:} The first version of AM implements a simplified version of the validation module which includes only DTD validation checks for each local application. (Implemented using the \texttt{xerces} Java parser [xml.apache.org]). In the next version we plan to add consistency checks (e.g. validity of request according to application state). Valid requests are passed to the request rewriter, otherwise a user notification is issued.

\textbf{Request rewriter and AM engine:} The two are combined together into one executable. The AM engine consists of a main program, which generates events according to the

\footnote{Other alternatives could be implementing it e.g. as a plug in for a user browser or as a software agent. We chose the above architecture for modularity reasons, but similar design principles could apply for the other configurations.}
accepted request, and a collection of modules which include the local application’s
state machines and are activated by those events. These modules are automatically
generated from the specification, as described below, using Rational’s *Rose RealTime*
tool [120] and include local State Charts definition as well as state machine activation
code. The Am engine communicate with each local application (using threads) via
the wrappers and writes the response to a file (a distinct XML file for each local
source.) These files are passed to the result integrator.

*Results integrator:* Written in Java and uses a DOM interface to read the xml local
data. The integrator reads and parses each file (for validity check and DOM tree cre-
ation), thus creating one tree per file, and then integrates them into one tree (using
the algorithm of Section 4.4.1).

### 4.5.2 Specification and Code Generation

For the specification of applications we use a UML compliant modeling tool - Ratio-
nal’s *Rose RealTime* tool - which provides a convenient graphical interface for writing
the specifications and automatic code generation from the specifications (plus debug-
gging and tracing facilities for the code).\(^5\) The generated modules realize the state
machines described in the specifications and are used by the AM engine as explained
above.

**Modeling Considerations and Structures** In section 4.2 we described the use
of State Charts to specify the activities, the methods and the flow. Recall that
activities corresponds to web pages while methods represents the operations that can
be activated using buttons within these pages. In our specification language, we have
used states to specify both methods and activities. As different activities (pages) can
be activated in parallel, they are described as orthogonal states according to [77]
(see Figure 4.5). Since our specification language follows the UML standard, the
system implementation could be simplified by using a UML compliant tool for the
specification and code generation. There are several such tools on the market (e.g. [82,

\(^5\)This in fact illustrates one of the main advantages of basing our framework on standards like
XML and UML - the ability to capitalize on enhanced tools developed for the standard.
We have chosen to use Rose RealTime as our implementation tool. This specific choice forced the implementation to deviate a bit from the specification language introduced in previous sections, to match the tool’s capabilities. Nevertheless, the changes are very minor and affect only the graphical syntax of the specification and not the semantics. We detail the changes below. Rose RealTime implementation of orthogonal states is based on the notion of capsule, which are a UML stereotype of a class. Therefore we use capsules to describe the activities, and their associated state chart to describe the methods and the flow. Figure 4.23 demonstrate the visual notation of capsules. Capsules are represented by rectangles, while the small black or white squares are ports which are used by capsules for a signal based communication (events transmission). Due to the use of capsules, the refinement process, used for specifying the local application, differs a little from the one described in Paragraph 4.3.2. First, local activities (capsules) are derived from global activities (capsules) by inheritance association. In this way, the local activity inherits the state chart (and sub capsules) of the corresponding global activity. From this point, we continue with the state chart refinement steps as described above.

**Modeling Example:** Now we return to our running example of a global bookstore which integrates services offered by several bookstores on the Web, and follow the modeling steps:

**Modeling the Global Application:** First we define the global data structure (see Figure 4.3 for the dtd), then we define the global application interface (Figure 4.25)
Figure 4.24: Local Application (with capsules)
which actually defines the API. Recall that forms and buttons correspond to data and methods. We create four pages. For each page we define an AM activity: Search, SearchResults, ShoppingCart and AddReviews. The search activity enable us to submit a query to multiple bookstores (SearchMethod) and browse the integrated catalog (BrowseMethod). (gotoBookPrices). On search activation, a new page appears, containing the search results. Each book description includes title, author,ISBN, list of the bookstores it was found in, price and availability details. The methods in this page are Add_to_Cart_method and goto_Add_Reviews. The next page shows the shopping cart’s content in all relevant bookstores and allows to delete_from_cart, change_quantity and initialize all carts. The last page (omitted from the figure) include an add_review method. Figure 4.23 is a partial UML representation of this interface in Rose RealTime.

Modeling Local Applications: Now we come to model the local applications. The UML modeling of the application flow and method is quite straightforward and derived from the local site interface. Once the global application is defined, it is used as a base class for defining of the local applications by inheritance association. Then the specific behavior is defined by refinement of the global State Charts. We also separate the handling of the communication from the modeling process. Calls to local sites are implemented in one separate general module, with one public method which accept the local sites parameters (requested site, methods and data to be passed) and supports issuing http calls and handling of cookies. In that way the code involved in activating a local site’s method within a state is minimal and merely includes calling the above method.

Figure 4.26 is a screenshot of three pages of a local bookstore. The specification of the corresponding state chart through refinement was explained in detail in Section 4.3 and illustrated in Figure 4.11. Figure 4.24 shows the equivalent UML model in Rose RealTime, using capsules. The definition of the DTD is quite simple. We can easily derive from the visual representation of the pages the DTD for Amazon’s local catalog. Each book item includes title, one or more authors, list price, Amazon’s price, availability etc. But sometimes there is additional hidden information. For example from looking at the html page source of the search results we can extract the
Figure 4.25: The Global Application Interface
ISBN of each book, which is a unique identifier for books and is important for data integration. After we have the local DTDs, we define the mapping between them and the global data as queries over the global data (see Figures 4.44.10).

**Code Generation:** Finally, after all this is completed, we can build the local bookstore module. We do this by creating a component which contains the local bookstore capsules, selecting the component in the browser, and clicking compile. *Rose Real-Time* then creates the make file based upon the platform, generates the C/C++ code for the application (based upon structure, state transition diagrams, and transition and state action logic), and initiates the C++ compiler to generate the code. Now that we have compiled, we can run the application from *Rose RealTime*, set breakpoints, animate the RoseRT model so that we can watch how the application is running. Then we deploy the dll file to the target machine to be activated by the AM Engine to communicate with the local applications.

### 4.5.3 Wrappers

A wrapper has two parts, the first in charge of the translation of the standard AM API requests to application-specific *http* calls, and the second in charge of mapping the returned data to the corresponding AM XML-based representation. The first direction can be implemented in two ways: One is rather straightforward and amounts to running an application Web browser in the background, mapping each API request to a corresponding action on the browser (typing or buttons activation). This is naturally facilitated by the fact that the local applications modeling is tied to the actual screens, hence the mapping is immediate. The second alternative is an actual formatting of corresponding *http* calls (with the appropriated parameters). While the first approach is applicable in most cases, the latter is possible only for *http*-based requests (e.g. not for Java rmi calls), and is feasible only when the correspondence between the API variables and the *http* call parameters is clear. Nevertheless, we decided to use the latter in our implementation: a proxy tracing of the *http* calls of the local bookstores showed a straightforward correspondence between the parameters hence writing the request translator was quite trivial. We plan to implement
Figure 4.26: Local Application Example
the former method in the next version of the system to support a wider range of applications. The request transformer is table-based perl script which reformatsthe general request according to the requested local site and method. In this version the table is stored as an XML file and will be managed in an XML repository in next versions. For each method in each local bookstore we need to add the an entry to this table. Figure 4.27 is an excerpt of this table. Notice that each entry defines the http request method (POST/GET), URL and the content to be sent incase of a POST action. Strings between exclamation marks are names of variables that should be extracted from the original XML request and replaced by their actual values.

The opposite direction, i.e. mapping html response to XML-based representation, requires parsing of the response. For that we used simple scripts written in perl that parse a specific page content and perform the above translation.

During the implementation we encountered the problem of web pages structure being periodically changed. We thus consider adopting methods for change detection and automatic wrappers development [87].

4.5.4 Run Time Operation

Now, after the specifications are defined and the appropriate wrappers are built, we will describe what happens at run time operation. Figure 4.28 illustrates the steps
that follows a user request: (1) The user activates one of the global application methods (e.g. search) by clicking on one of the global application interface buttons. (2) The interface via the user’s browser issues a request to the AM system (Recall that the AM system is implemented as a CGI program and runs under a web server). For example Figure 4.29 demonstrates a request for books search to all sites that support such method. The AM parses the request, decompose it to several requests (e.g. one for each bookstore), validates and rewrites them. Each request is then handled (in a different thread) by the appropriate local store’s module as part of the AM engine. These modules send one request or a sequence of requests, according to the local application flow, via the proxy server (3), which is in charge of wrappers activation. These requests are a site specific request in a unified format and contain local application’s cookies and variables maintained by the AM engine. The proxy server activates a wrapper, which transforms the request according to an xml file (Figure 4.27) to the specific format of the local application (see Figure 4.30) and
4.6. A POSTERIORI PERSPECTIVE TO THE AM

4.6 A Posteriori Perspective to the AM

We now consider integration and composition of Web services in perspective of previous chapters. As mentioned above, this work was done in 2000, before Web Services standards emerged. Nowadays, additional approaches based on Web Services may be taken. In this section, we sketch a possible implementation of the AM paradigm, following the lines of the BP-QL data model. We compare the two solutions and discuss the pros and cons of each approach.

As shown above, the specification of an AM application has two parts. The first part models the global application, which is the target integrated/customized application, while the second describes the local Web applications and their relationship to the global one. In this solution we assume that the underlying (local) applications export their services as Web services, and provide their descriptions using the WSDL standard [130]. We can use BPEL to describe the global applications, and BPEL refinements to describe how global activities can be implemented by each local application’s activities. The XML wrappers described above, can be easily replaced with new wrappers that provide Web Service support over the existing HTML applications.

The main drawback of using BPEL comparing to AM is the lack of support for
data integration. Although the BPEL standard, supports the declarative specification of composing Web services into Business Processes, using complex flows and parallelism, it has no support of declarative definition for data integration. Data mappings between different formats of different components are written by hand using Xpath and embedded in the BPEL operations. In the implementation that was done a few years ago, modeling the local data variables as views (queries) over the global variables is also written by hand. This process can be automated using current technology of automatic schema matching, out of which the mapping between the local variables and the global variables can be derived (see for example [66, 99]). Then the queries can be embedded as Xpath or XSLT [130] transformations in the BPEL specifications.

We can see that the AM approach can be easily modified to use the BPEL standard. BPEL refinements are very similar to State Charts refinements and goes well with the AM approach of modeling the local applications as views of the global application. BPEL specifications provide similar capabilities including concurrency and the ability to activate remote components, although the usage may involve some additional performance overhead.

4.7 Summary

The AM system tackles the problem of integrating and customizing existing Web applications. Based on XML and UML it offers a novel solution, integral approach with declarative specifications dealing with both data and flow Semantics. We use maximal view rewriting for mapping the data, and State Charts refinement for mapping the flow. The system supports a declarative specification language for specifying the integration and customization task. Then, acting as an application generator, it generates a full integrated/customized e-commerce application.
Chapter 5

Related work

According to the three parts of the thesis, we discuss related work, and consider our results in perspective of state of the art in these areas.

5.1 The BP Model and Query Language

We presented BP-QL, a novel graphical Query Language for querying Business Processes. BP-QL allows users to query business processes visually, in a manner very close to how such processes are typically specified, and can be employed in a distributed P2P setting. We described the formal model underlying the BP-QL query language, studied the properties of the language components, and explained how they influenced the language design. We have also described the system implementation, highlighting the main challenges faced and the solutions taken.

The BP-QL language is based on an intuitive model of business processes, an abstraction of the emerging BPEL (Business Process Execution Language) standard [23]. Other previously proposed standards like [60, 25, 48] can similarly be supported, exploiting the abstraction level of our formal model.

Process models Several concrete models have been suggested for describing Web Services behavior including state-machine [21], PI calculus [134], a variant of Mealy state machine using process algebra for transitions [79], Petri-nets [127], models to
trace-based formalisms like Message Sequence Charts [32]. BPEL4WS language includes artifacts from both XLANG [134] and WSFL [92], each of which took a different approach to workflow. Basically, there are two main approaches; reactive models like State Charts vs. interactive models like Petri-nets. Web services are reactive components, activated by a message arrival via their interface, and as such it is best to describe their behavior using a reactive model.

**Program analysis and verification** There has been a vast amount of previous work in the general area of program analysis and verification (see e.g. [70, 88] for a sample), and more specifically in the analysis of interactions of composite web services and BPEL processes [58, 70, 52]. These works mostly consider logic-based query languages where queries, formulated as logic formulas, test if the runs of the application or program satisfies a certain property; a witness counter example is provided if not. In contrast, we advocate here an intuitive, visual query formulation, where queries are written in essentially the same way as process specifications. **BP-QL** allows not only to test if a certain pattern occurs, but also displays to the user all the relevant paths. Indeed a major contribution of the present work is the construction of a concise finite representation of the (possibly infinite) set of results.

As mentioned in Section 2.1, program verification is typically of very high complexity (from NP-hard for very simple specifications to undecidable in the general case [107, 70].) To guaranty complexity that is polynomial in the size of the data, **BP-QL** queries process specifications, rather than possible runs, ignoring the run-time semantics of certain BPEL constructs such as ‘choice’, parallel execution, and variable values. Identifying semantic constructs that can nevertheless be incorporated without increasing complexity is a challenging future research. It is also interesting to study whether certain data structures (e.g. BDD [88]) that are used to speed up program verification tasks can also be employed in our context to further accelerate query evaluation.

**Software specification query languages** The importance of software specifications query languages for understanding and restructuring of software systems has bee recognized. Query-based approaches have been used for software re-engineering,
code refactoring and improving software evolution and quality [126, 85].

Various formats and query languages are used to store and retrieve various aspects of the programs. For example, [94] uses relations to represent syntactic and semantic data, and SQL to query them. [44] represents structural information in a Prolog database, and a graphical query language to identify and remove cyclic control dependencies. [117] describes an algebra that can be used to express relational queries on source text. [126] formulates a query algebra to express direct queries on the program syntax tree.

In the context of model-driven software development, the Object Management Group (OMG) proposes a standard for a UML [110] transformation language [111] that includes a query language based on simple object patterns. [93] presents a query language for UML (and other modeling languages). The query language uses graph pattern matching and supports repetitive structures and negation. The ideas of the latter are in a sense similar to our language, i.e. queries are written in a similar way to the spec. However, this language is targeted in general modeling languages, in contrast to BP-QL that supports requirements that are specific to BPEL like flexible granularity, paths extraction and more.

Visual query languages The design of BP-QL was inspired by previous works on visual query languages for XML, such as XML-GL [40] and XQBE [26]. These languages are descendants of a long line of research on graph based query languages such as G [46], Graphlog [43] and G-Log [114]. The main innovation of BP-QL is in introducing process patterns that enrich the standard path-based navigation with (1) a (transitive) zoom-in, that allows to query process components at any depth of nesting, and (2) the retrieval of paths of interest. Together, these features allow for simple formulation of queries on BPs, but also make the evaluation of queries more intricate than that of flat graphs. To keep the evaluation of queries tractable, we had identified the problematic scenarios and carefully designed the language so that they are avoided, and polynomial-time query evaluation is guaranteed. Extending the language to allow also for the construction of new processes based on the retrieved data is an on-going research.
The importance of BP Query language  The importance of query languages for business processes has been recognized by BPMI (the Business Process Management Initiative) who started a BPQL (Business Processes Query Language) initiative in 2002 [24]. However, no draft standard was published since. We hope that BP-QL will contribute to such a standard. Complementary to our work is the research performed in the area of Business Process Management (BPM) and Business Process Intelligence (BPI). Both academic (e.g., [122, 30, 128] and commercial tools (e.g., [15, 78, 81]) have been developed to support the definition, execution, and monitoring of BPs, including systems for extracting knowledge from event logs (process mining). BP-Mon extends BP-QL to serve as a basis for a general query platform, that allows queries that involve process specifications as well as execution data.

5.2 Querying Data Streams

In chapter 3 we presented BP-Mon, a novel query language for monitoring BPs. BP-Mon allows to design complex monitoring tasks that deal with both events and flow; it offers easy, user-friendly design of such tasks; and it compiles these tasks into standard BPEL processes, thus providing easy deployment, portability, and minimal overhead. We now discuss some of the language design and implementation challenges with respect to related work.

Composite events  BP monitoring entails the detection and processing of composite events. Event detection is at the core of several related application domains, such as active databases, publish-subscribe and production systems[132, 72, 122]. A variety of formalisms have been proposed for the specification of composite events, including event algebras, situation calculus, temporal languages, process algebra, transaction logic and computation tree logic (see [115] for an overview), allowing to define composite events based on the time stamps and casual dependencies of individual (or other composite) events. As explained in the Introduction, a key difference of the present work is the higher abstraction level employed here. Following the BPEL philosophy, BP-Mon users need not be aware of the underlying implementation details
of the monitored BP and the type of run-time events generated by the system. The specifications of monitoring tasks is performed on the same (high) level of abstraction as that of the BPs specification.

**Runtime monitoring**  A vast amount of work was performed on verification of concurrent and distributed systems using specification languages such as LTL. Recently, this approach has been applied to Web service composition, using the BPEL framework [84]. Runtime monitoring based on LTL, Statecharts, and related formalisms has also received a lot of attention recently (see [121] and [102]). These works are mainly focused on error detection, e.g. concurrency related bugs.

Intrusion detection systems also analyze audit trails to find bad scenarios, but they mostly perform string pattern matching in the bit/ character level to search for anomalies and attack signatures [103].

Our approach is different in that it relies on the BPEL model for the visual language used to specify monitoring requests, and on execution on a BPEL engine for the monitoring itself. These points contribute to the ease of deployment and the efficiency of execution of our monitoring tool.

**Querying data streams**  XML and relational stream processing has received much attention in the database and systems research, with application domains such as sensor networks, network traffic, financial analysis, transaction log analysis, and auctioning [73]). Several projects [106, 29, 36] assume a relational data model and propose SQL-like syntax and enhanced support for windows and ordering. Works in the area of active data bases propose supporting procedural continuous queries with event-condition-action rules (e.g. [72]). Projects like [98] also provide SQL-like syntax, with object-oriented data model. Several XML query engines, e.g. [68, 54, 83, 96, 118], focus on optimizing the pattern retrieval in XML queries. The events in a BP trace are given in XML but have additional semantics. We have already mentioned in Section 3.1 the limitations of existing XML engines here. To overcome them, BP-Mon supports flow patterns, as opposed to XML patterns, and uses a dedicated pattern matching algorithm.
**DFA vs. NFA** Many XML filtering engines are based on automata, either Deterministic Finite-state Automata (DFA) or Non-deterministic ones (NFA). Early implementations use one NFA for each path query [8]. Later works [54, 35] support path sharing, converting large numbers of XPath queries into a single NFA. [74, 75] are based on DFA, and handle a limited subset of XPath Boolean queries. NFA-based approaches are space efficient, requiring a relatively small number of states to represent complex queries. DFA-based approaches are time efficient since their state transitions are deterministic, but the conversion from an NFA to a DFA increases the number of states exponentially. To avoid this exponential blow up, works like [74] compute the states lazily, at run-time. Our work is similar. We use a DFA, auto-generated as a BPEL process, which instantiates the required states (activities) as it progresses.

**Memory requirements** Lower bounds on the space required for the evaluation of continuous select-project-join queries over relational streams are considered in [11]. The authors showed that queries require a potentially unbounded amount of memory unless the domains of the attributes involved in the query are restricted. [13] presents a lower bound technique, which given a query, specifies the minimum amount of memory required for its evaluation. The challenges encountered in our work are similar, but the queries are inherently more complex. Our queries and data are (nested) DAGs; DAG-shaped pattern entails join operations. Investigating syntactic restrictions on EX-patterns and BP specifications to provide lower bounds for memory needs is an interesting question.

**Optimization** A variety of methods have been proposed for optimized processing of (composite) events, employing relation and object-oriented database technology [132, 72], petri nets, finite state automata, event graphs, and storage minimization (See [116] for a survey). These methods are generic, that is they can be employed in a variety of application domains. To our knowledge the present work is the first to propose a BP-specific optimization that exploits knowledge about the BP structure, and is complementary to the above works. Optimizing multiple queries for concurrent evaluation is important in the context of resource sharing in continuous queries (see
5.3. PROCESS INTEGRATION

In our implementation, we employ a simple variant of this idea, factorizing the pattern matching of multiple report points attached to one EX-pattern. The use of schema knowledge is another important XML query optimization technique [68, 54, 75]. In our case, the “schema” is the BP specification and the optimization goal is to reduce computation time and memory requirements for the BPEL processes implementing the queries. We presented an optimization along these lines. Further optimization may include pattern simplifications, e.g. replacing non-transitive edges with transitive ones and reducing pattern nesting by eliminating unnecessary compound activities.

BP management In Section 3.1 we reviewed BAM (Business Activity Monitoring) systems. As mentioned, typical BAM systems are targeted at enterprise applications. Software runtime analysis tools like Purify, Quantify and PureCoverage [80] closely follow the execution of applications and allow to create extensive reports about memory usage, memory leaks, memory and performance bottlenecks, and code coverage. These tools are very low level and are targeted at developers. We enrich the context of BP monitoring by flow patterns, flexible granularity, intuitive visual interface, and enable ad hoc querying and fast deployment, that goes along with the dynamic nature of Web services.

Complementary to this line of work is the post-analysis of traces that were gathered and stored in databases [31]. Some works mine logs to predict execution metrics like activity completion time, process duration, and exceptions, while others use data mining to derive the BP structure. Extending BP-Mon with facilities for querying stored logs is an on-going research.

5.3 Process Integration

The Application Manifold data model and specification language are inspired by [1], where a system called ActiveViews was proposed for the generation of new e-commerce applications. But the goal of our AM specifications (hence also the semantics of the specifications and type of the corresponding generated applications) is completely
different: while ActiveViews specifications were used for constructing new e-commerce applications from scratch, the AM system tackles the integration and customization of existing Web applications.

**Software Integration vs. Data Integration** In recent years, significant research effort has focused on software composition. Early work, consider components composition and reuse in the context of object-oriented programming [69, 109]). A higher level approach, more suitable for the composition of Web applications, is Megaprogramming [133]. For instance, the CHAIMS project [34], handles the composition of remote services, typically provided by autonomous suppliers. As in AM, the architecture is based on a repository of services including locations, protocols and data types, and of wrappers for services not supporting the CHAIMS protocol. Extensive work has been also done in the context of the new web services standards (a detailed survey can be found in [119]). For example, [6] describes a composition methodology based on semantically annotating web service, [101] extends a BPEL engine to support runtime semantic service selection, and [124] proposed automatic BPELs composition based on goal specification and a model checking problem. But the focus of work in this area is mainly on the client’s developing environment, automatic composition, parallel execution, scheduling, and parameter handling, while the actual integration of data manipulated by the modules and the profile of the associated e-commerce applications are mostly ignored. Similarly, the BPEL standard itself, supports the declarative specification of composing Web services into Business Processes, using complex flows and parallelism, but has no support of declarative definition for data integration. Data mapping between the different formats is written by hand in Xpath and embedded in the BPEL operations.

Data integration has been the focus of much recent research [27, 37, 28, 90, 71, 105, 49]. However, the emphasis was on querying the data available in Web applications. The application flow, when considered, is mostly limited to the navigation process needed for extracting dynamic web content (data extracted by filling multiple forms) [49]. Following these lines, many commercial Web sites, like comparative shopping services, that use the above technology, are not much more than search services; when
shoppers wish to perform a purchase, they are directed to the merchant web sites (e.g. [42, 50]). Some sites do provide purchase capabilities (e.g. [137, 9]) but these are often based on pre-agreement with the merchants, using hard-coded interfaces or maintaining merchant catalogs, as opposed to our system which generates a full integrated Web application in a transparent way that does not necessarily demand the source site’s cooperation. An exception is [12] that enables both product search and purchase in multiple sites, and which was released about the same time as our prototype. But since no information is available regarding the internal structure of the product, one cannot tell if it the underlying technology is dedicated specifically to comparative shopping or provides a generic support for application composition.

[33] offers a visual language for modeling data intensive Web sites, and a set of CASE tools that enables automatic code generation. The specifications include the data, the pages, and the navigation, presentation and personalization models. Although the system supports integration of legacy data sources, it doesn’t handle the full scope of application integration, e.g. integration of data from different sources into one data entity, or concurrent activation of different flows to perform an integrated action that involves several sites.

GAV, LAV and GLAV The AM paradigm is an adaptation of the Information Manifold (IM) approach [90], used for data integration, to the context of e-commerce applications integration. In IM, local data is modeled as a traditional view of the global data. A significant difference here is that AM views cover the full profile of e-commerce applications, not just the data.

Specifically, there are three main formalisms proposed for specifying semantic mappings between multiple sources [89, 100]. In the first, called global-as-view (GAV) [71], the global data is described as a view (composition) over the local data. In the second, local-as-view (LAV) that is used in our system, the local sources are described as views over the global schema. In the third approach, global-local-as-view (GLAV) [65, 76], the relationships between the global schema and the sources make use of both LAV and GAV assertions. Specifically, every assertion has the form $q_S(X) \subseteq q_G(X)$, where $q_S$ is a conjunctive query over the local source, and $q_G$ a
conjunctive query over the global schema.

We have chosen the LAV approach for its suitability for modular application
development, where the integration of new local applications into an existing system
is rather convenient, requiring only the specification of the source view in terms of
the global application and, when needed, the supply of an appropriate wrapper for
the source. This is important in our context since new e-commerce applications are
added to the Web every day and existing applications are often modified. Reducing
the amount of work involved in incorporating new applications or adjusting to changes
is thus an important factor.

It should also be noted that, in any case, no matter which specification direction
is chosen (i.e. local as view or global as view), since the information flows in both
ways (local data is integrated and passed to the global application, while updated
global data and user input is sliced and distributed among the local applications),
the inverse mapping will be needed as well. In our case this is generated automati-
cally - AM introduces a novel rewriting paradigm, enhancing the traditional maximal
view rewriting with a refinement mechanism dealing with the operational aspects
of the application. Also, for the data part, most of the previous work on maximal
view rewriting has been in the context of relational data. Works dealing with more
complex data models, like semi-structured and OODB data, considered only exact
query rewriting [97] (rather than maximal) and query containment [91]. A contribu-
tion here is the consideration of maximal view rewriting in the context of XML data,
providing executable queries to combine data from several XML sources. Later works
consider query containment for XPath expressions [104, 53], while [55] addresses con-
tainment of XML queries with nesting and propose a set of algorithms for XML query
containment, minimization and mapping composition.
Chapter 6

Conclusion and Future Work

This thesis addresses several challenges pertaining to querying, monitoring and integrating Business Processes. We argued that the new BP standards not only simplify software development, but, more interestingly from an information management perspective, they also provide an important new mine of information. Queries about the BPs, that were extremely hard (if not impossible) to evaluate when the BP logic was coded in complex programs are now potentially much easier given a declarative specification. Furthermore, sophisticated querying, that interleaves static analysis of the BP specification with run-time process monitoring, can now be used for a variety of critical tasks such as fraud detection, SLA (service level agreement) maintenance, and general business management. This provides an essential infrastructure for companies to optimize business processes, reduce operational costs, and ultimately increase competitiveness.

In Chapter 2 we presented BP-QL, a novel graphical query language for querying Business Processes. BP-QL allows users to query business processes visually, in a manner very close to how such processes are typically specified, and can be employed in a distributed P2P setting. We described the formal model underlying the BP-QL query language, studied the properties of the language components, and explained how they influenced the language design. We have also described the system implementation, highlighting the main challenges faced and the solutions taken.

In Chapter 3 we also presented BP-Mon, a novel query language that extends the


BP-QL platform to monitor business processes executions using a query language. BP-Mon allows to design complex monitoring tasks that deal with both events and flow; it offers easy, user-friendly design of such tasks; and it compiles these tasks into standard BPEL processes, thus providing easy deployment, portability, and minimal overhead.

Finally, in Chapter 4 we presented the Application Manifold system which, based on the Web standards XML and UML, offers a novel solution for specifying the integration and customization task of Web applications, covering the full profile of the integrated/customized e-commerce applications: the various services offered by the applications, the activities, the application flow, as well as the data involved in the process. Then, acting as an application generator, the system generates a full integrated/customized e-commerce application.

The work presented in this thesis lays the basis for a platform to handle various aspects of Business Processes. There are many interesting future directions to look at.

6.1 Querying BP Specifications

Partial information In this work we have assumed that all relevant specifications are available for our queries. As mentioned above, Business Processes may belong to different organizations, and some of specifications may not be publicly available. An interesting research challenge is how to answer questions in such a context.

Applications to software verification We have mentioned that in querying the specifications with BP-QL, we are capturing only part of the BPEL semantics, ignoring certain BPEL constructs such as conditional execution and variable values and focuses on the given specification flow. This is a reasonable tradeoff to guarantee complexity that is polynomial in the size of the data, since the verification problem of querying the possible runs of a system is typically of very high complexity (from NP-hard for very simple specifications to undecidable in the general case.

It is interesting to investigate how this method of querying of specifications can be
applied to “approximates” verification queries. Even when full verification is desired, BP-QL can be used as an efficient means to narrow the search space for the more costly, interpretation dependent, verification.

Additional semantics As mentioned in Chapter 2, from complexity considerations we ignore semantics of certain BPEL constructs and data values. This is a reasonable tradeoff between expressivity and complexity. It is to see what happens if we extend BP-QL with additional BPEL semantics.

Optimization In BP-QL, we relied on the optimizations incorporated in AXML (like parallel access and lazy retrieval) to provide good performance. Additional optimization can be achieved by incorporate caching mechanism for already retrieved data, and by using graph indexes (e.g. [39], [123]).

6.2 Monitoring BP Executions

Partial information As opposed to specifications, when there is a clear benefit for organizations to expose the specifications, it is reasonable to assume that in most cases the they will not expose their BP executions to monitoring, or will restrict monitoring to certain events. An interesting research challenge is how to answer questions when parts of the executions are missing. One possible direction is doing some estimations regarding the missing parts according to their outgoing messages.

Optimization In BP-Mon we used an optimization technique that prunes redundant monitoring, based on inconsistency and irrelevance. The main goals of such optimizations is to reduce the number of the events that need to be processes and maintained in memory, and reducing the number of backtracking. Further optimization may include pattern simplifications, e.g. replacing non-transitive edges with transitive ones and reducing pattern nesting by eliminating unnecessary compound activities.

Optimizing multiple queries for concurrent evaluation is also important in the context of resource sharing. In our implementation, we employ a simple optimization
of factorizing the pattern matching of multiple report points attached to one EX-pattern. The study of further factorization is an interesting future work.

Another interesting research direction is, investigating syntactic restrictions on EX-patterns and BP specifications to provide lower bounds for memory needs.

**Querying logs**  This work is focused on querying the specifications and monitoring systems at runtime. The main difference in querying logs is that in the latter we will be interesting in retrieving all possible answers and not just a first match. Querying the logs should deal with large amount of data, and the compact representation of the results. Extending BP-Mon with facilities for querying stored logs is an on-going research.

**6.3 Integrating BPs**

We presented the *Application Manifold* that simplify the task of integrating and customizing http-based Web applications, and showed how this system can be modified to use current BPEL technology. There are integration and composition challenges whether we are dealing with Web applications or BPEL processes that can be addresses by using a query language. Discovering the desired services that fit certain data and flow requirements, extracting the flow that need to be supported, automatic generation of the glue logic are just a few examples. We hope that, as query languages are used for data integration, BPQL will assist in process integration.
Bibliography


[34] CHAIMS. Compiling high-level access interfaces for multi-site software. http://www-db.stanford.edu/CHAIMS/.


[80] IBM software - Rational Purify.


[105] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In Int. Conf. on very Large DataBases (VLDB), New-York, August 1998.


ytshav shel inor wainTEGRITZA Shel Yishomi Gel

hiybor leshem kbolat hoveir "Doktor Leفلוסופיה"

mat

unat ayil

ubedda zo nusheha bhadretam shva

prof. toveh miloa vprof. cahiriel bairi

nogeq leshamat shlo oinibritat teli-avbi

agoose 2007
тяжית

עמוס עברוב

לגווייל במספר בפסנתר של ישומי, Web, ישן יותר ויוור הדגמי עובד לבליט שרותלים הדתים על

סימן מיום קים. לאמשל קוטבי ורשבאיים, קנייה שלווהית תונה. אוול מגוון המונחראטיים ויהיו

והשתתת בחיי, בתרות ורשבאיים, (flow) במדים של אחרים ורשבאיים (מקשMiami) זא

על התשומת של אינטגרציה ומקומדיצציה של ישומי אלח. פתרון אינטגרציה ומקומדיצציה ו_tA

ירקינם, תצלום לעת מענה רק לשבכי השתדנה של מחצר אלקטרונים.احتנה זו המספר בשמה

סטנדרטים הפתרונות חלך מבזות הקיימים, אך על מעבר לא היה פתרון סטנדרטי לבניית התהליכים

עסקיים.

acrof תמצית

, (Business Process Execution Language) BPEL - ב

את המקומ בושק המסגרת הד-פקוט, המאפרשת למקף, SHRITOM (Web Services) ממקף

הסמספקיותרוקדולציטות לשן BPEL ממקף לשן פיתוח התנוהוות ממקף, ממקף לשון

גנון, את ממקף更多 לשון ממקף. ויוול לשון ממקף ממקף, השם אוול לגנון והו

לאשימניף פריסי, ממקף לאשימני ליתשתה התהליכים והסמספקים אוול, למקף. כמך ממקף

של התהליכים כאשר אוול לשוני ממקף עלが出דה, אוול צירית למקף ממקף ס留守ה עכיזיה, размещен אחזרת.

BP-Mon -BP-QL

cדי למקף משולחת שארנה, או ממקף בובוזד ושת שת שיאלוהי.

אות השפה שארנה (Business Process Query Language) BP-QL

景德 תמצית לשון של jap תמצית לשון של jap

טסמי. השיפוף המסומסר על מקול אינטגראזרי של התהליכים טסמי, שאותו שנספה

הסמספקיותרוקדולציטות לשן BPEL הסמספקיותרוקדולציטות לשן BPEL

המקף יבשות להמקף במקף במקף, שנות רכיב התהליכים הסמספקים לע

ארוגנימ שנון. ממקף מתמקף ממקף במקף פטרונה, במקף בצומת הכריה אינטגראזרי של

BP-QL ממקף מתמקף יבשות לשון במקף במקף פטרונה, במקף בצומת הכריה אינטגראזרי של

טסמי. ממקף מתמקף יבשות לשון במקף במקף פטרונה, במקף בצומת הכריה אינטגראזרי של

למקף物联网 אינטגראזרי של الجهاز, למקף物联网 אינטגראזרי של الجهاز, שנות מחפש ממקף

מקף, וממקף, שנות מחפש ממקף

שקול תצעיפ עיתונה בוצעה שלמח ממקפים בפוסם התיהה.

במקף物联网 אינטגראזרי של الجهاز, או פתרון, הר-AM

(HTML Application Manifold) momentarily "לשתות שארה ורשוםTextNode שחשيمي, HTML, א продолжа

את תמצית לעת מתמקף במקף במקף, במקף בצומת הכריה אינטגראזרי של ישומי, במקף במקף, במקף

לאינטגרציה לשון תוצות, או פתרון דרכ כל המקף物联网 אינטגראזרי של ישומי, במקף במקף, במקף

לבניית ישומי והרשימה, במקף物联网 אינטגראזרי של ישומי, במקף物联网 אינטגראזרי של ישומי, במקף物联网 אינטגראזרי של ישומי.
The research on Web services aims to develop and promote Web Services (UDDI, XML, SOAP, WSDL) that can be used in various applications and use cases. The focus is on creating standards that are independent of specific vendors, such as Microsoft, IBM, Oracle, SAP, Siebel, BEA, and others. The research is driven by the need for interoperability and the ability to share information and services across different platforms and technologies.

The BPEL (Business Process Execution Language) standard is used to represent business processes, allowing for the automatic composition of services. BPEL provides a language for describing business processes in a way that is independent of the underlying implementation.

The use of BPEL allows for the creation of services that can be invoked by other services, providing a flexible and scalable way to build complex systems. The use of standards such as WSDL and XML allows for the creation of services that can be accessed by different applications, regardless of the underlying technology.

The research on Web services is also driven by the need for standards that are flexible and can adapt to changing requirements. The use of standards such as WSFL and XLANG allows for the creation of services that can be easily modified and adapted to changing needs.

The research on Web services is also driven by the need for standards that can be used to integrate different systems and applications. The use of standards such as UDDI and WSDL allows for the creation of directories that can be used to discover and access different services.

The use of standards such as BPEL and the ability to write business processes in a language that is independent of the underlying implementation allows for the creation of services that can be used in a variety of different environments, including the cloud.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily managed and monitored. The use of standards such as UDDI and the ability to discover and manage services allows for the creation of services that can be easily scaled and managed.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily tested and validated. The use of standards such as WSDL and XML allows for the creation of services that can be easily tested and validated, ensuring that they meet the requirements of different environments and applications.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily integrated with other systems and applications. The use of standards such as UDDI and WSDL allows for the creation of services that can be easily integrated with other systems and applications, ensuring that they can be used in a variety of different environments and applications.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily managed and monitored. The use of standards such as UDDI and the ability to discover and manage services allows for the creation of services that can be easily scaled and managed.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily tested and validated. The use of standards such as WSDL and XML allows for the creation of services that can be easily tested and validated, ensuring that they meet the requirements of different environments and applications.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily integrated with other systems and applications. The use of standards such as UDDI and WSDL allows for the creation of services that can be easily integrated with other systems and applications, ensuring that they can be used in a variety of different environments and applications.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily managed and monitored. The use of standards such as UDDI and the ability to discover and manage services allows for the creation of services that can be easily scaled and managed.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily tested and validated. The use of standards such as WSDL and XML allows for the creation of services that can be easily tested and validated, ensuring that they meet the requirements of different environments and applications.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily integrated with other systems and applications. The use of standards such as UDDI and WSDL allows for the creation of services that can be easily integrated with other systems and applications, ensuring that they can be used in a variety of different environments and applications.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily managed and monitored. The use of standards such as UDDI and the ability to discover and manage services allows for the creation of services that can be easily scaled and managed.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily tested and validated. The use of standards such as WSDL and XML allows for the creation of services that can be easily tested and validated, ensuring that they meet the requirements of different environments and applications.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily integrated with other systems and applications. The use of standards such as UDDI and WSDL allows for the creation of services that can be easily integrated with other systems and applications, ensuring that they can be used in a variety of different environments and applications.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily managed and monitored. The use of standards such as UDDI and the ability to discover and manage services allows for the creation of services that can be easily scaled and managed.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily tested and validated. The use of standards such as WSDL and XML allows for the creation of services that can be easily tested and validated, ensuring that they meet the requirements of different environments and applications.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily integrated with other systems and applications. The use of standards such as UDDI and WSDL allows for the creation of services that can be easily integrated with other systems and applications, ensuring that they can be used in a variety of different environments and applications.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily managed and monitored. The use of standards such as UDDI and the ability to discover and manage services allows for the creation of services that can be easily scaled and managed.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily tested and validated. The use of standards such as WSDL and XML allows for the creation of services that can be easily tested and validated, ensuring that they meet the requirements of different environments and applications.

The research on Web services is also driven by the need for standards that can be used to develop and deploy services that can be easily integrated with other systems and applications. The use of standards such as UDDI and WSDL allows for the creation of services that can be easily integrated with other systems and applications, ensuring that they can be used in a variety of different environments and applications.
ה أمسפים את קדמתם של תהליכים עקיף היכולת או לתשובות בקווים כללי, בייעדים ה쉽אות נוכל להרצת אתר באופנים דומים, כי אםIGINיים唐山חרי Denied שמתל幕墙 ב коллектив ומקודדת למדים על קווים בעלת ספסיפיקציות של ל_preferences, העבר ואיים עיקריות של תהליכים ושפתossip הפרטניים היישומים של שאילתות, ושפתossip המסרוע עיקריות של תהליכים עד ליישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלニーוסיישומים של שאילתות, שאילתות המדינה שלנייוסיישומים של שאילתות, שאילתות המדינה שלנייוסיישומים של שאילתות, שאילתות המדינה שלנייוסיישומים של שאילתות, שאילתות המדינה שלנייוסיישומים של שאילתות, שאילתות המדינה שלנייוסיישומים של שאילתות, שאילתות המדינה שלנייוסיישומים של שאילתות, שאילתות המדינה שלנייוסיישומים של שאילתות, שא={<パートסא愛情}}
The document discusses the integration of new applications over the Web, enabling the search and purchase of products through various means, such as electronic shops of similar products that offer various services. The system also allows the booking of flights, virtual travel agencies, and car rental services. The last new issue is a group of applications that offer facilities to support convenient tools to provide here is our goal. We must find a solution to the issue of the various transactions of the integration, these flows (and activities), the data of the integration and these are.

This work presents the results and we will conclude below, and the challenges we address here, and the issues described in chapter 2.

The languages of the question marks and the specifications are written in a way that makes it possible to draw graphs of their activities, and these are used in the various maps, and the language is also used for zoom-in.

The language of the languages of the languages is used for the language of the languages, and this is also used for the language of the languages, and the language is also used for zoom-in.

The language of the languages is used for the language of the languages, and this is also used for the language of the languages, and the language is also used for zoom-in.

The language of the languages is used for the language of the languages, and this is also used for the language of the languages, and the language is also used for zoom-in.


A filter for extracted conclusions (activity) (activity). The activity time of the first event (the second event is the activity time of the second event) is the time of the first event (the second event is the activity time of the second event).

